

СВЕТЛИН НАКОВ  
& КОЛЕКТИВ



Python

# ОСНОВИ НА ПРОГРАМИРАНЕТО С **Python**



SoftUni  
Foundation



Software  
University



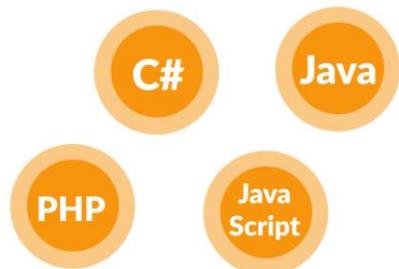
# Кратко съдържание

Кратко съдържание .....	3
Съдържание .....	7
Предговор .....	13
Глава 1. Първи стъпки в програмирането.....	27
Глава 2.1. Прости пресмятания с числа.....	61
Глава 2.2. Прости пресмятания с числа – изпитни задачи.....	95
Глава 3.1. Прости проверки.....	111
Глава 3.2. Прости проверки – изпитни задачи.....	133
Глава 4.1. По-сложни проверки .....	147
Глава 4.2. По-сложни проверки – изпитни задачи.....	165
Глава 5.1. Повторения (цикли) .....	183
Глава 5.2. Повторения (цикли) – изпитни задачи .....	201
Глава 6.1. Вложени цикли.....	217
Глава 6.2. Вложени цикли – изпитни задачи .....	237
Глава 7.1. По-сложни цикли.....	249
Глава 7.2. По-сложни цикли – изпитни задачи .....	281
Глава 8.1. Подготовка за практически изпит – част I .....	293
Глава 8.2. Подготовка за практически изпит – част II .....	317
Глава 9.1. Задачи за шампиони – част I.....	333
Глава 9.2. Задачи за шампиони – част II.....	347
Глава 10. Функции.....	363
Глава 11. Хитрости и хакове .....	389
Заключение.....	401

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Основи на програмирането с Python

Светлин Наков и колектив

Бончо Вълков

Венцислав Петров

Владимир Дамяновски

Илия Илиев

Йордан Даракчиев

Мартин Царев

Миглен Евлогиев

Милена Ангелова

Мирела Дамянова

Николай Костов

Петър Иванов

Петя Господинова

Светлин Наков

Таня Евтимова

Таня Станева

Теодор Куртев

Христо Минков

ISBN: 978-619-00-0806-4

София, 2018

# Основи на програмирането с Python

© Светлин Наков и колектив, 2018 г.

Първо издание, последна редакция: октомври 2018 г.

Настоящата книга се разпространява **свободно** под **отворен лиценз CC-BY-NC-SA**, който определя следните права и задължения:

- **Споделяне** – можете да копирате и разпространявате книгата свободно във всякакви формати и медиии.
- **Адаптиране** – можете да копирате, миксирате и променяте части от книгата и да създавате нови материали на базата на извадки от нея.
- **Признание** – при използване на извадки от книгата трябва да цитирате оригиналния източник, настоящия лиценз и да опишете направените промени, без да въвеждате потребителя в заблуда, че оригиналните автори подкрепят вашата работа.
- **Некомерсиална употреба** – нямате право да използвате книгата за комерсиални цели.
- **Подобно споделяне** – ако създавате материали чрез миксиране, промяна и копиране на извадки от книгата, трябва да споделите резултата под същия или подобен лиценз.

Всички запазени марки, използвани в тази книга, са собственост на техните притежатели.

Издателство: Фабер, гр. Велико Търново

ISBN: 978-619-00-0806-4

Корица: Марина Шидерова – <http://shideroff.com>

Официален уеб сайт: <https://python-book.softuni.bg>

Официална Facebook страница: <https://fb.com/IntroProgrammingBooks>

Сурс код: <https://github.com/SoftUni/Programming-Basics-Book-Python-BG>

# Съдържание

Кратко съдържание .....	3
Съдържание .....	7
Предговор .....	13
За кого е тази книга? .....	13
Защо избрахме езика Python? .....	14
Книгата на други програмни езици: C#, Java, C++, PHP .....	14
Програмиране се учи с много писане, не с четене!.....	14
За Софтуерния университет (Софтуерни).....	15
Как се става програмист? .....	17
Книгата в помощ на учителите.....	22
Историята на тази книга.....	23
Официален сайт на книгата .....	24
Форум за вашите въпроси.....	24
Официална Facebook страница на книгата.....	25
Лиценз и разпространение .....	25
Докладване на грешки .....	25
Приятно четене!	25
<b>Глава 1. Първи стъпки в програмирането.....</b>	<b>27</b>
Видео .....	27
Какво означава "да програмираме"? .....	27
Python интерпретатор: инсталация и използване .....	31
Компютърни програми.....	33
Как да напишем конзолна програма? .....	34
Среда за разработка (IDE) .....	35
Пример: създаване на конзолна програма "Hello Python" .....	39
Тествайте програмите за свирене на ноти .....	42
Типични грешки в Python програмите .....	43
Какво научихме от тази глава?.....	43
Упражнения: първи стъпки в програмирането.....	44
Конзолни, графични и уеб приложения .....	49
Упражнения: графични и уеб приложения .....	49
<b>Глава 2.1. Прости пресмятания с числа.....</b>	<b>61</b>
Видео .....	61
Пресмятания в програмирането .....	61
Типове данни и променливи .....	61
Печатане на резултат на екрана .....	62
Четене на потребителски вход – цяло число .....	62
Четене на поредица числа .....	63
Четене на дробно число .....	64
Четене на вход под формата на текст .....	64

Съединяване на текст и числа .....	65
Аритметични операции .....	66
Съединяване на текст и число .....	68
Отпечатване на форматиран текст в Python .....	69
Числени изрази .....	72
Закръгляне на числа .....	73
Какво научихме от тази глава? .....	75
Упражнения: прости пресмятания .....	75
Графични приложения с числови изрази .....	87
<b>Глава 2.2. Прости пресмятания с числа – изпитни задачи .....</b>	<b>95</b>
Четене на числа от конзолата .....	95
Извеждане на текст по шаблон (placeholder) .....	95
Аритметични оператори .....	96
Конкатенация .....	96
Изпитни задачи .....	97
Задача: учебна зала .....	97
Задача: зеленчукова борса .....	100
Задача: ремонт на плочки .....	102
Задача: парички .....	105
Задача: дневна печалба .....	108
<b>Глава 3.1. Прости проверки .....</b>	<b>111</b>
Видео .....	111
Сравняване на числа .....	111
Прости проверки .....	112
Проверки с if-else конструкция .....	113
За блоковете от код .....	113
Живот на променлива .....	115
Серии от проверки .....	116
Упражнения: прости проверки .....	117
Дебъгване - прости операции с дебъгер .....	120
Упражнения: прости проверки .....	122
<b>Глава 3.2. Прости проверки – изпитни задачи .....</b>	<b>133</b>
Изпитни задачи .....	133
Задача: цена за транспорт .....	133
Задача: тръби в басейн .....	136
Задача: поспаливата котка Том .....	138
Задача: реколта .....	141
Задача: фирма .....	143
<b>Глава 4.1. По-сложни проверки .....</b>	<b>147</b>
Видео .....	147
Вложени проверки .....	147
По-сложни проверки .....	150

---

Логическо "И" .....	150
Логическо "ИЛИ" .....	152
Логическо отрицание .....	153
Операторът скоби () .....	154
По-сложни логически условия .....	154
Какво научихме от тази глава? .....	159
Упражнения: по-сложни проверки .....	159
<b>Глава 4.2. По-сложни проверки – изпитни задачи.....</b>	<b>165</b>
Вложени проверки .....	165
if-elif-else проверки .....	165
Изпитни задачи .....	165
Задача: навреме за изпит .....	166
Задача: пътешествие.....	169
Задача: операции между числа.....	173
Задача: билети за мач.....	176
Задача: хотелска стая.....	179
<b>Глава 5.1. Повторения (цикли).....</b>	<b>183</b>
Видео .....	183
Повторения на блокове код (for цикъл).....	183
Какво научихме от тази глава? .....	191
Упражнения: повторения (цикли) .....	191
Упражнения: графични и уеб приложения .....	194
<b>Глава 5.2. Повторения (цикли) – изпитни задачи .....</b>	<b>201</b>
Изпитни задачи .....	201
Задача: хистограма .....	201
Задача: умната Лили .....	205
Задача: завръщане в миналото .....	208
Задача: болница.....	210
Задача: деление без остатък .....	213
Задача: логистика .....	215
<b>Глава 6.1. Вложени цикли.....</b>	<b>217</b>
Видео .....	217
Вложени цикли .....	218
Чертане на по-сложни фигури .....	223
Какво научихме от тази глава? .....	230
Упражнения: чертане на фигурки в уеб среда.....	230
<b>Глава 6.2. Вложени цикли – изпитни задачи .....</b>	<b>237</b>
Изпитни задачи .....	237
Задача: чертане на крепост .....	237
Задача: пеперуда .....	239
Задача: знак "Стоп" .....	241

Задача: стрелка.....	243
Задача: брадва .....	245
<b>Глава 7.1. По-сложни цикли.....</b>	<b>249</b>
Видео.....	249
Цикли със стъпка.....	249
While цикъл.....	251
Най-голям общ делител (НОД) .....	253
Алгоритъм на Евклид.....	253
While True + break цикъл .....	254
Безкрайни цикли и операторът break.....	257
Вложени цикли и операторът break.....	260
Справяне с изключения: try-except .....	261
Задачи с цикли.....	263
Какво научихме от тази глава? .....	267
Упражнения: уеб приложения с по-сложни цикли.....	268
<b>Глава 7.2. По-сложни цикли – изпитни задачи .....</b>	<b>281</b>
Изпитни задачи .....	281
Задача: генератор за тъпи пароли .....	281
Задача: магически числа .....	283
Задача: спиращо число .....	286
Задача: специални числа .....	288
Задача: цифри .....	290
<b>Глава 8.1. Подготовка за практически изпит – част I.....</b>	<b>293</b>
Видео.....	293
Практически изпит по “Основи на програмирането” .....	293
Система за онлайн оценяване (Judge).....	293
Задачи с прости пресмятания.....	293
Задачи с единична проверка .....	297
Задачи с по-сложни проверки .....	300
Задачи с единичен цикъл .....	304
Задачи за чертане на фигурки на конзолата .....	308
Задачи с вложени цикли с по-сложна логика .....	312
<b>Глава 8.2. Подготовка за практически изпит – част II.....</b>	<b>317</b>
Видео.....	317
Изпитни задачи .....	317
Задача: разстояние.....	317
Задача: смяна на плочки .....	320
Задача: магазин за цветя.....	322
Задача: оценки .....	325
Задача: коледна шапка .....	327
Задача: комбинации от букви .....	330

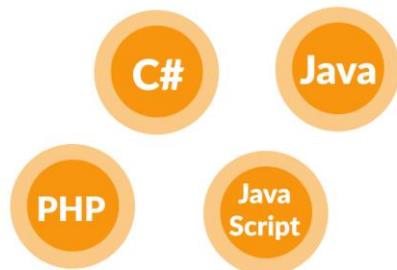
---

<b>Глава 9.1. Задачи за шампиони – част I.....</b>	<b>333</b>
По-сложни задачи върху изучавания материал.....	333
Задача: пресичащи се редици .....	333
Задача: магически дати.....	337
Задача: пет специални букви.....	341
<b>Глава 9.2. Задачи за шампиони – част II.....</b>	<b>347</b>
По-сложни задачи върху изучавания материал.....	347
Задача: дни за страстно пазаруване.....	347
Задача: числен израз .....	352
Задача: бикове и крави .....	356
<b>Глава 10. Функции.....</b>	<b>363</b>
Видео .....	363
Какво е функция? .....	363
Функции с параметри (по-сложни функции).....	367
Връщане на резултат от функции.....	371
Варианти на функции.....	375
Вложени функции (локални функции).....	376
Утвърдени практики при работа с функции .....	377
Какво научихме от тази глава? .....	379
Упражнения .....	380
<b>Глава 11. Хитрости и хакове .....</b>	<b>389</b>
Форматиране на кода .....	389
Именуване на елементите на кода.....	391
Бързи клавиши в PyCharm .....	392
Шаблони с код (code snippets).....	393
Техники за дебъгване на кода .....	397
Справочник с хитрости .....	398
Какво научихме от тази глава? .....	400
<b>Заключение.....</b>	<b>401</b>
Тази книга е само първа стъпка! .....	401
Накъде да продължим след тази книга? .....	402
Онлайн общности за стартиращите в програмирането .....	405
Успех на всички! .....	406

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Предговор

Книгата "Основи на програмирането" е официален учебник за курса "Programming Basics" за начинаещи в Софтуерния университет (СофтуУни): <https://softuni.bg/courses/programming-basics>. Тя запознава читателите с писането на програмен код на начално ниво (basic coding skills), работа със среда за разработка (IDE), използване на променливи и данни, оператори и изрази, работа с конзолата (четене на входни данни и печтане на резултати), използване на условни конструкции (**if, if-else**), цикли (**for, while**) и работа с функции (деклариране и извикване на функции, подаване на параметри и връщане на стойност). Използват се езикът за програмиране **Python** и средата за разработка **PyCharm**. Обхванатият учебен материал дава базова подготовка за по-задълбочено изучаване на програмирането и подготвя читателите за приемния изпит в СофтуУни.



Тази книга ви дава само **първите стъпки към програмирането**. Тя обхваща съвсем начални умения, които предстои да развивате години наред, докато достигнете до ниво, достатъчно за започване на работа като програмист.

Книгата се използва и като неофициален [учебник за училищните курсове по програмиране в професионалните гимназии](#) в България, изучаващи професиите "Програмист", "Приложен програмист" и "Системен програмист", както и като допълнително учебно пособие в началните курсове по програмиране в **средните училища, профилираните и математическите гимназии**, за паралелките с профил "информатика и информационни технологии".

## За кого е тази книга?

Тази книга е подходяща за **напълно начинаещи** в програмирането, които искат да опитат какво е да програмираш и да научат основните конструкции за създаване на програмен код, които се използват в софтуерната разработка, независимо от езиците за програмиране и използваните технологии. Книгата дава една **солидна основа** от практически умения, които се използват за по-нататъшно обучение в програмирането и разработката на софтуер.

За всички, които не са преминали [бесплатния курс по основи на програмирането за напълно начинаещи в СофтуУни](#), специално препоръчваме да го запишат **напълно бесплатно**, защото програмиране се учи с правене, не с четене! На курса ще получите бесплатно достъп до учебни занятия, обяснения и демонстрации на живо или онлайн (като видео уроци), **много практика и писане на код**, помощ при решаване на задачите след всяка тема, достъп до преподаватели, асистенти и ментори, както и форум и дискусионни групи за въпроси, достъп до общност от хиляди навлизащи в програмирането и всякаква друга помощ за начинаещия.

Бесплатният курс в СофтуУни за напълно начинаещи е подходящ за **ученици** (от 5 клас нагоре), **студенти** и **работещи** други професии, които искат да натрупат технически знания и да разберат дали им харесва да програмират и дали биха се занимавали сериозно с разработка на софтуер за напред.

Нова група започва всеки месец. Курсът "Programming Basics" в СофтУни се организира регулярно с няколко различни езика за програмиране, така че опитайте. Обучението е **безплатно** и може да се откажете по всяко време, ако не ви допадне. **Записването** за бесплатно присъствено или онлайн обучение за стаптиращи в програмирането е достъпно през **формата за кандидатстване в СофтУни**: <https://softuni.bg/apply>.

## Защо избрахме езика Python?

За настоящата книга избрахме езика **Python**, защото е **съвременен език** за програмиране от високо ниво и същевременно е лесен за научаване и **подходящ за начинаещи**. Като употреба **Python** е **широкоразпространен**, с добре развита екосистема, с многобройни библиотеки и технологични рамки и съответно дава много **перспективи** за развитие. **Python** комбинира парадигмите на процедурното, функционалното и обектно-ориентираното програмиране по съвременен начин с лесен за употреба синтаксис. В книгата ще използваме **езика Python** и средата за разработка **PyCharm**, която е достъпна бесплатно от JetBrains.

Както ще обясним по-късно, **езикът за програмиране, с който стаптираме, няма съществено значение**, но все пак трябва да ползваме някакъв програмен език, и в тази книга сме избрали именно **Python**. Книгата може да се намери преведена огледално и на други езици за програмиране като C#, Java, JavaScript и C++.

## Книгата на други програмни езици: C#, Java, C++, PHP

Настоящата книга по програмиране за напълно начинаещи е достъпна на няколко езика за програмиране (или е в процес на адаптация за тях):

- [Основи на програмирането със C#](#)
- [Основи на програмирането с Java](#)
- [Основи на програмирането с JavaScript](#)
- [Основи на програмирането с Python](#)
- [Основи на програмирането със C++](#)
- [Основи на програмирането с PHP](#)

Ако предпочитате друг език, изберете си от списъка по-горе.

## Програмиране се учи с много писане, не с четене!

Ако някой си мисли, че ще прочете една книга и ще се научи да програмира без да пише код и да решава здраво задачи, определено е в заблуда. Програмирането се учи с **много, много практика**, с писане на код всеки ден и решаване на стотици, дори хиляди задачи, сериозно и с постоянство, в продължение на години.

Трябва **да решавате здраво задачи**, да бъркате, да се поправяте, да търсите решения и информация в Интернет, да пробвате, да експериментирате, да намирате по-добри решения, да свиквате с кода, със синтаксиса, с езика за

програмиране, със средата за разработка, с търсенето на грешки и дебъгването на неработещ код, с разсъжденията над задачите, с алгоритмичното мислене, с разбиването на проблемите на стъпки и имплементацията на всяка стъпка, да трупате опит и да вдигате уменията си всеки ден, защото да се научиш да пишеш код е само **първата стъпка към професията "софтуерен инженер"**. Имате да учите много, наистина много!

Съветваме читателя като минимум да пробва всички примери от книгата, да си поиграе с тях, да ги променя и тества. Още по-важни от примерите, обаче, са **задачите за упражнения**, защото те развиват практическите умения.

**Решавайте всички задачи от книгата**, защото програмиране се учи с практика! Задачите след всяка тема са внимателно подбрани, така че да покриват в дълбочина обхванатия учебен материал, а целта на решаването на всички задачи от всички обхванати теми е да дадат **цялостни умения за писане на програмен код** на начално ниво (каквато е целта и на тази книга). На курсовете в СофтУни не случайко **наблягаме на практиката** и решаването на задачи, и в повечето курсове писането на код в клас е над 70% от целия курс.



**Решавайте всички задачи за упражнения от книгата.** Иначе нищо няма да научите! Програмиране се учи с писане на много код и решаване на хиляди задачи!

## За Софтуерния университет (Софтуни)

Софтуерният университет (Софтуни) е най-мащабният учебен център за софтуерни инженери в България. През него преминават десетки хиляди студенти всяка година. СофтУни отваря врати през 2014 г. като продължение на усилията на д-р Светлин Наков масирано да изгражда **кадърни софтуерни специалисти** чрез истинско, съвременно и качествено образование, което комбинира фундаментални знания със съвременни софтуерни технологии и много практика.

Софтуерният университет предоставя **качествено образование, професия, работа и възможност за придобиване на бакалавърска степен** за програмисти, софтуерни инженери и ИТ специалисти. СофтУни изгражда изключително успешно трайна връзка между **образование и индустрия**, като си сътрудничи със стотици софтуерни фирми, осигурява **работа и стажове** на своите студенти, предоставя качествени специалисти за софтуерната индустрия и директно отговаря на нуждите на работодателите чрез учебния процес.

## Бесплатните курсове по програмиране в СофтУни

Софтуни организира **бесплатни курсове по програмиране** за напълно начинаещи в цяла България - присъствено и онлайн. Целта е всеки, който има **интерес** към програмиране и технологии, да опита **програмирането** и да се увери сам дали то е интересно за него и дали иска да се занимава сериозно с разработка на софтуер.

Можете да се запишете за **бесплатния курс по основи на програмирането** от страницата за кандидатстване в СофтУни: <https://softuni.bg/apply>.

Бесплатните курсове по основи на програмирането в СофтУни имат за цел да ви запознаят с **основните програмни конструкции** от света на софтуерната разработка, които ще можете да приложите при всеки един език за програмиране. Те включват работа с **данни, променливи и изрази**, използване на **проверки, конструиране на цикли** и дефиниране и извикване на **функции** и други похвати за изграждане на програмна логика. Обученията са **изключително практически насочени**, което означава, че **силно се набляга на упражнения**, а вие получавате възможността да приложите знанията си още докато ги усвоявате.

Настоящият **учебник по програмиране** съпътства бесплатните курсове по програмиране за начинаещи в СофтУни и служи като допълнително учебно помагало, в помощ на учебния процес.

## Judge системата за проверка на задачите

Софтуни Judge системата (<https://judge.softuni.bg>) представлява автоматизирана система в Интернет за проверка на решения на задачи по програмиране чрез **поредица от тестове**. Предаването и проверката на задачите се извършва в **реално време**: пращате решение и след секунди получавате отговор дали е вярно. Всеки **успешно** преминат тест дава предвидените за него точки. При вярно решение получавате всички точки за задачата. При частично вярно решение получавате част от точките за дадената задача. При напълно грешно решение, получавате 0 точки.

**Всички задачи от настоящата книга са достъпни за тестване в СофтУни Judge системата** и силно препоръчваме да ги тествате, след като ги решите, за да знаете дали не изпускате нещо и дали наистина решението ви работи правилно, според изискванията на задачата.

Имайте предвид и някои **особености на SoftUni Judge системата**:

- За всяка задача Judge системата пази **най-високия постигнат резултат**. Ако качите решение с грешен код или по-слаб резултат от предишното ви изпратено, системата няма да ви отнеме точки.
- Изходните резултати на вашата програма се **сравняват** от системата стриктно с очаквания резултат. Всеки **излишен символ, липсваща запетайка или интервал** може доведе до 0 точки на съответния тест. **Изходът**, който Judge системата очаква, е **описан в условието на всяка задача** и към него не трябва да се добавя нищо повече.

**Пример:** ако в изхода се изисква да се отпечата число (напр. **25**), не извеждайте описателни съобщения като **The result is: 25**, а отпечатайте точно каквото се изисква, т.е. само числото.

Софтуни Judge системата е **достъпна по всяко време** от нейния сайт: <https://judge.softuni.bg>.

- За вход използвайте автентикацията си от сайта на СофтУни: <https://softuni.bg>.
- Използването на системата е **бесплатно** и не е обвързано с участието в курсовете на СофтУни.

Убедени сме, че след няколко изпратени задачи, ще ви хареса да получавате **моментална обратна връзка** дали написаното от вас решение е вярно, и Judge системата ще ви стане най-любимия помощник при учене на програмирането.

## Как се става програмист?

Драги читатели, сигурно много от вас имат амбицията да стават програмисти, да си изкарват прехраната с разработка на софтуер или да работят в ИТ сектора. Затова сме пригответи за вас **кратко ръководство "Как се става програмист"**, за да ви ориентираме за стъпките към тази така желана професия.

Програмист (на ниво започване на работа в софтуерна фирма) се става за **най-малко 1-2 години здраво учене и писане на код всеки ден**, решаване на няколко хиляди задачи по програмиране, разработка на няколко по-сериозни практически проекта и трупане на много опит с писането на код и разработката на софтуер. Не става за един месец, нито за два! Професията на софтуерния инженер изисква голям обем познания, покрити с много, много практика.

Има **4 основни групи умения**, които всички програмисти трябва да притежават. Повечето от тези умения са устойчиви във времето и не се влияят съществено от развитието на конкретните технологии (които се променят постоянно). Това са уменията, които **всеки добър програмист притежава** и към които всеки новобранец трябва да се стреми:

- писане на код (20%)
- алгоритмично мислене (30%)
- фундаментални знания за професията (25%)
- езици и технологии за разработка (25%)

## Умение #1 – кодене (20%)

Да се научите **да пишете код** формира около 20% от минималните умения на програмиста, необходими за започване на работа в софтуерна фирма. Умението да кодиш включва следните компоненти:

- работа с променливи, проверки, цикли
- ползване на функции, методи, класове и обекти
- работа с данни: масиви, списъци, хеш-таблици, стрингове

Умението да кодиш **може да се усвои за няколко месеца** усилено учене и здраво решаване на практически задачи с писане на код всеки ден. Настоящата книга покрива само първата точка от умението да кодиш: **работка с променливи**,

**проверки и цикли.** Останалото остава да се научи в последващи обучения, курсове и книги.

Книгата (и курсовете, базирани на нея) дават само началото от едно дълго и сериозно учене, по пътя на професионалното програмиране. Ако не усвоите до съвършенство учебния материал от настоящата книга, няма как да станете програмист. Ще ви липсват фундаментални основи и ще ви става все по-трудно напред. Затова **отделете достатъчно внимание на основите на програмирането:** решавайте здраво задачи и пишете много код месеци наред, докато се научите да решавате с лекота всички задачи от тази книга. Тогава продължете напред.

Специално обръщаме внимание, че **езикът за програмиране няма съществено значение** за умението да кодиш. Или можеш да кодиш или не. Ако можеш да кодиш на **Python**, лесно ще се научиш да кодиш и на Java, и на C++, и на друг език. Затова **уменията да кодираш** се изучават доста сериозно в началните курсове за софтуерни инженери в СофтУни (вж. [учебния план](#)) и с тях стартира всяка книга за програмиране за напълно начинаещи, включително нашата.

## Умение #2 – алгоритмично мислене (30%)

Алгоритмичното (логическо, инженерно, математическо, абстрактно) мислене формира около 30% от минималните умения на програмиста за старт в професията. **Алгоритмичното мислене** е умението да разбивате една задача на логическа последователност от стъпки (алгоритъм) и да намирате решение за всяка отделна стъпка, след което да сглобявате стъпките в работещо решение на първоначалната задача. Това е най-важното умение на програмиста.

Как да си изградим алгоритмично мислене?

- Алгоритмичното мислене се развива се чрез решаване на **много (1000+)** **задачи** по програмиране, възможно най-разнообразни. Това е рецептата: решаване на хиляди практически задачи, измисляне на алгоритъм за тях и имплементиране на алгоритъма, заедно с дебъгване на грешките по пътя.
- Помагат физика, математика и/или подобни науки, но не са задължителни! Хората с **инженерни и технически наклонности** обикновено по-лесно се научават да мислят логически, защото имат вече изградени умения за решаване на проблеми, макар и не алгоритмични.
- Способността **да решавате задачи по програмиране** (за която е нужно алгоритмично мислене) е изключително важна за програмиста. Много фирми изпитват единствено това умение при интервюта за работа.

Настоящата книга развива **начално ниво на алгоритмично мислене**, но съвсем не е достатъчна, за да ви направи добър програмист. За да станете кадърни в професията, ще трябва да добавите **умения за логическо мислене и решаване на задачи** отвъд обхвата на тази книга, например работа със **структурни от данни** (масиви, списъци, матрици, хеш-таблици, дърворидни структури) и базови **алгоритми** (търсене, сортиране, обхождане на дърворидни структури, рекурсия и други).

**Умения за алгоритмично мислене** се развиват сериозно в началните курсове за софтуерни инженери в СофтУни (вж. [учебния план](#)), както и в специализираните курсове по [структурни от данни](#) и [алгоритми](#).

Както може би се досещате, **езикът за програмиране няма значение** за развирането на алгоритмичното мислене. Да мислиш логически е универсално, дори не е свързано само с програмирането. Именно заради силно развитото логическото мислене се счита, че **програмистите са доста умни** и че прост човек не може да стане програмист.

## Умение #3 – фундаментални знания за професията (25%)

Фундаменталните знания и **умения** за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки формират около 25% от минималните умения на програмиста за започване на работа. Ето по-важните от тези знания и умения:

- **базови математически концепции**, свързани с програмирането: координатни системи, вектори и матрици, дискретни и недискретни математически функции, крайни автомати и state machines, понятия от комбинаториката и статистика, сложност на алгоритъм, математическо моделиране и други
- **умения да програмираш** - писане на код, работа с данни, ползване на условни конструкции и цикли, работа с масиви, списъци и асоциативни масиви, стрингове и текстообработка, работа с потоци и файлове, ползване на програмни интерфейси (APIs), работа с дебъгер и други
- **страници от данни и алгоритми** - списъци, дървета, хеш-таблици, графи, търсене, сортиране, рекурсия, обхождане на дърворидни структури и други
- **обектно-ориентирано програмиране (ООП)** – работа с класове, обекти, наследяване, полиморфизъм, абстракция, интерфейси, капсуляция на данни, управление на изключения, шаблони за дизайн
- **функционално програмиране (ФП)** - работа с ламбда функции, функции от по-висок ред, функции, които връщат като резултат функция, затваряне на състояние във функция (closure) и други
- **бази данни** - релационни и нерелационни бази данни, моделиране на бази данни (таблици и връзки между тях), език за заявки SQL, технологии за обектно-релационен достъп до данни (ORM), транзакционност и управление на транзакции
- **мрежово програмиране** - мрежови протоколи, мрежова комуникация, TCP/IP, понятия, инструменти и технологии от компютърните мрежи
- взаимодействие **клиент-сървър**, комуникация между системи, back-end технологии, front-end технологии, MVC архитектури
- **технологии за сървърна (back-end) разработка** - архитектура на уеб сървър, HTTP протокол, MVC архитектура, REST архитектура, frameworks за уеб разработка, templating engines

- **уеб front-end технологии (клиентска разработка)** - HTML, CSS, JS, HTTP, DOM, AJAX, комуникация с back-end, извикване на REST API, front-end frameworks, базови дизайн и UX (user experience) концепции
- **мобилни технологии** - мобилни приложения, Android и iOS разработка, мобилен потребителски интерфейс (UI), извикване на сървърна логика
- **вградени системи** - микроконтролери, управление на цифров и аналогов вход и изход, достъп до сензори, управление на периферия
- **операционни системи** - работа с операционни системи (Linux, Windows и други), инсталация, конфигурация и базова системна администрация, работа с процеси, памет, файлова система, потребители, многозадачност, виртуализация и контейнери
- **паралелно програмиране и асинхронност** - управление на нишки, асинхронни задачи, promises, общи ресурси и синхронизация на достъпа
- **софтуерно инженерство** - сурс контрол системи, управление на разработката, планиране и управление на задачи, методологии за софтуерна разработка, софтуерни изисквания и прототипи, софтуерен дизайн, софтуерни архитектури, софтуерна документация
- **софтуерно тестване** - компонентно тестване (unit testing), test-driven development, QA инженерство, докладване на грешки и трекери за грешки, автоматизация на тестването, билд процеси и непрекъсната интеграция

Трябва да поясним и този път, че **езикът за програмиране няма значение** за усвояването на всички тези умения. Те се натрупват бавно, в течение на много години практика в професията. Някои знания са фундаментални и могат да се усвояват теоретично, но за пълното им разбиране и осъзнаването им в дълбочина, са необходими години практика.

Фундаментални знания и умения за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки се учат по време на [цялостната програма за софтуерни инженери в СофтУни](#), както и с редица [изборни курсове](#). Работата с разнообразни софтуерни библиотеки, програмни интерфейси (APIs), технологични рамки (frameworks) и софтуерни технологии и тяхното взаимодействие, постепенно изграждат тези знания и умения, така че не очаквайте да ги добиете от единичен курс, книга или проект.

За започване на работа като програмист обикновено са достатъчни само **начални познания в изброените по-горе области**, а задълбаването става на работното място според използваните технологии и инструменти за разработка в съответната фирма и екип.

## Умение #4 - езици за програмиране и софтуерни технологии (25%)

Езиците за програмиране и технологиите за софтуерна разработка формират около 25% от минималните умения на програмиста. Те са най-обемни за

научаване, но най-бързо се променят с времето. Ако погледнем **обявите за работа** от софтуерната индустрия, там често се споменават всякачки думички (като изброените по-долу), но всъщност в обявите мълчаливо **се подразбират първите три умения**: да кодиш, да мислиш алгоритично и да владееш фундамента на компютърните науки и софтуерното инженерство.

За тези чисто технологични умения вече **езикът за програмиране има значение**.

- **Обърнете внимание:** само за тези 25% от професията има значение езикът за програмиране!
- **За останалите 75% от уменията няма значение езикът** и тези умения са устойчиви във времето и преносими между различните езици и технологии.

Ето и някои често използвани езици и технологии (software development stacks), които се търсят от софтуерните фирми (актуални към януари 2018 г.):

- **JavaScript** (JS) + ООП + ФП + бази данни + MongoDB или MySQL + HTTP + уеб програмиране + HTML + CSS + DOM + jQuery + Node.js + Express + Angular или React
- **C#** + ООП + ФП + класовете от .NET + база данни SQL Server + Entity Framework (EF) + ASP.NET MVC + HTTP + HTML + CSS + JS + DOM + jQuery
- **Java** + Java API classes + ООП + ФП + бази данни + MySQL + HTTP + уеб програмиране + HTML + CSS + JS + DOM + jQuery + JSP/Servlets + Spring MVC или Java EE / JSF
- **PHP** + ООП + бази данни + MySQL + HTTP + уеб програмиране + HTML + CSS + JS + DOM + jQuery + Laravel / Symfony / друг MVC framework за PHP
- **Python** + ООП + ФП + бази данни + MongoDB или MySQL + HTTP + уеб програмиране + HTML + CSS + JS + DOM + jQuery + Django
- **C++** + ООП + STL + Boost + native development + бази данни + HTTP + други езици
- **Swift** + macOS + iOS + Cocoa + Cocoa Touch + XCode + HTTP + REST + други

Ако изброените по-горе думички ви изглеждат страшни и абсолютно непонятни, значи сте съвсем в началото на кариерата си и имате **да учате още години** докато достигнете професията "**софтуерен инженер**".

Не се притеснявайте, всеки програмист преминава през един или няколко технологични стека и се налага да изучи **съвкупност от взаимосвързани технологии**, но в основата на всичко това стои **умението да пишеш програмна логика** (**да кодиш**), което се развива в тази книга, и **умението да мислиш алгоритично** (да решаваш задачи по програмиране). Без тях не може!

## **Езикът за програмиране няма значение!**

Както вече стана ясно, **разликата между езиците за програмиране** и по-точно между уменията на програмистите на различните езици и технологии, е в около **10-20% от уменията**.

- Всички програмисти имат около **80-90% еднакви умения**, които не зависят от езика! Това са уменията да програмираш и да разработваш софтуер, които са много подобни в различните езици за програмиране и технологии за разработка.
- Колкото повече езици и технологии владеете, толкова по-бързо ще учените нови и толкова по-малко ще усещате разлика между тях.

Наистина, **езикът за програмиране почти няма съществено значение**, просто трябва да се научите да програмирате, а това започва с **коденето** (настоящата книга), продължава в по-сложните **концепции от програмирането** (като структури от данни, алгоритми, ООП и ФП) и включва усвояването на **фундаментални знания и умения за разработка на софтуер, софтуерно инженерство и компютърни науки**.

Едва накрая, когато захванете конкретни технологии в даден софтуерен проект, ще ви трябват **конкретен език за програмиране**, познания за конкретни програмни библиотеки (APIs), работни рамки (frameworks) и софтуерни технологии (front-end UI технологии, back-end технологии, ORM технологии и други). Спокойно, ще ги научите, всички програмисти ги научават, но първо се научават на фундамента: **да програмират и то добре**.

Настоящата книга използва езика **Python**, но той не е съществен и може да се замени с Java, C#, JavaScript, PHP, C++, Ruby, Swift, Go, Kotlin или друг език. За овладяване на **професията "софтуерен разработчик"** е необходимо да се научите да **кодите** (20%), да се научите да **мислите алгоритично** и да решавате **проблеми** (30%), да имате **фундаментални знания по програмиране и компютърни науки** (25%) и да владеете **конкретен език за програмиране и технологиите около него** (25%). Имайте търпение, за година-две всичко това може да се овладеет на добро начално ниво, стига да сте сериозни и усърдни.

## Книгата в помощ на учителите

Ако сте **учител по програмиране, информатика или информационни технологии** или искате **да преподавате програмиране**, тази книга ви дава нещо повече от добре структуриран учебен материал с много примери и задачи. **Бесплатно** към книгата получавате **качествено учебно съдържание** за преподаване в училище, на **български език**, съобразено с училищните изисквания:

- Учебни презентации (PowerPoint слайдове) за всяка една учебна тема, съобразени с 45-минутните часове в училищата – бесплатно.
- Добре разработени задачи за упражнения в клас и за домашно, с детайлно описани условия и примерен вход и изход – бесплатно.
- Система за автоматизирана проверка на задачите и домашните (Online Judge System), която да се използва от учениците, също бесплатно.
- Видео-уроци с методически указания от **бесплатния курс за учители по програмиране**, който се провежда регулярно от СофтУни фондацията.

Всички тези **безплатни преподавателски ресурси** можете да намерите на сайта на СофтУни фондацията, заедно с учебно съдържание за цяла поредица от курсове по програмиране и софтуерни технологии. Изтеглете ги свободно от тук: <http://softuni.foundation/projects/applied-software-developer-profession>.

## Историята на тази книга

Главен двигател и ръководител на проекта за създаване на настоящата **свободна книга по програмиране за начинаещи** с отворен код е [д-р Светлин Наков](#). Той е основен идеолог и създател на учебното съдържание от [курса "Основи на програмирането" в СофтУни](#), който е използван за основа на книгата.

Всичко започва с масовите **безплатни курсове по основи на програмирането**, провеждани в цялата страна от 2014 г. насам, когато стартира инициативата "Софтуни". В началото тези курсове имат по-голям обхват и включват повече теория, но през 2016 г. д-р Светлин Наков изцяло ги преработва, обновява, опростява и **насочва много силно към практиката**. Така е създадено ядрото на **учебното съдържание от тази книга**.

Безплатните обучения на СофтУни за старт в програмирането са може би най-мащабните, провеждани някога в България. До 2018 г. курсът на СофтУни по основи на програмирането **се провежда над 200 пъти в близо 40 български града** присъствено и многократно онлайн, с над 70 000 участника. Съвсем естествено възниква и нуждата да се напише **учебник** за десетките хиляди участници в курсовете на СофтУни по програмиране за начинаещи. На принципа на свободния софтуер и свободното знание, Светлин Наков повежда **екип от доброволци** и задвижва този open-source проект, първоначално за създаване на книга по основи на програмирането с езика C#, а по-късно и с други езици за програмиране.

Проектът е част от усилията на [Фондация "Софтуерен университет"](#) да създава и разпространява отворено учебно съдържание за обучение на софтуерни инженери и ИТ специалисти.

## Авторски колектив

Настоящата книга "Основи на програмирането с Python" е разработена от широк авторски колектив от **доброволци**, които отделиха от своето време, за да ви подарят тези систематизирани знания и насоки при старта в програмирането. Списък на всички автори и редактори (по азбучен ред):

Бончо Вълков, Венцислав Петров, Владимир Дамяновски, Илия Илиев, Йордан Даракчиев, Мартин Царев, Миглен Евлогиев, Милена Ангелова, Мирела Дамянова, Николай Костов, Петър Иванов, Петя Господинова, Светлин Наков, Таня Евтимова, Таня Станева, Теодор Куртев, Христо Минков

Книгата е базирана на нейния първоначален C# вариант (Въведение в програмирането със C#: <https://csharp-book.softuni.bg>), който е разработен от широк авторски колектив и който има принос и към настоящата книга:

Александър Кръстев, Александър Лазаров, Ангел Димитриев, Васко Викторов, Венцислав Петров, Даниел Цветков, Димитър Татарски, Димо Димов, Диян Тончев, Елена Роглева, Живко Недялков, Жулиета Атанасова, Захария Пехливанова, Ивелин Кирилов, Искра Николова, Калин Примов, Кристиян Памидов, Любослав Любенов, Николай Банкин, Николай Димов, Павлин Петков, Петър Иванов, Росица Ненова, Руслан Филипов, Светлин Наков, Стефка Василева, Теодор Куртев, Тоньо Желев, Християн Христов, Христо Христов, Цветан Илиев, Юlian Линев, Яница Вълева

Дизайн на корица: Марина Шидерова – <https://linkedin.com/in/marina-shideroff>.

Книгата е написана в периода юни-октомври 2018.

## Официален сайт на книгата

Настоящата книга по **Основи на програмирането с Python** за начинаещи е достъпна за свободно ползване в Интернет от адрес:

<https://python-book.softuni.bg>

Това е **официалният сайт на книгата** и там ще бъде качвана нейната последна версия. Книгата е преведена огледално и на други езици за програмиране, посочени на нейния сайт.

## Форум за вашите въпроси

Задавайте вашите въпроси към **настоящата книга** по основи на програмирането във форума за технически дискусии на СофтУни:

<https://softuni.bg/forum>

В този дискусионен форум ще получите безплатно **адекватен отговор по всякакви въпроси от учебното съдържание на настоящия учебник**, както и по други въпроси от програмирането. Общността на СофтУни за навлизачи в програмирането е толкова голяма, че обикновено отговор на зададен въпрос се получава **до няколко минути**. Преподавателите, асистентите и менторите от СофтУни също отговарят постоянно на вашите въпроси.

Поради големия брой учащи по настоящия учебник, във форума можете да намерите **решение на практически всяка задача от него**, споделено от ваш колега. Хиляди студенти преди вас вече са решавали същите задачи, така че ако закъснате, потърсете из форума. Макар и задачите в курса "Основи на програмирането" да се сменят от време на време, споделянето е винаги наследявано в СофтУни и затова лесно ще намерите решения и насоки за всички задачи.

Ако все пак имате конкретен въпрос, например защо не тръгва дадена програма, над която умувате от няколко часа, **задайте го във форума** и ще получите отговор. Ще се учудите колко добронамерени и отзивчиви са обитателите на СофтУни форума.

## Официална Facebook страница на книгата

Книгата си има и **официална Facebook страница**, от която може да следите за новини около книгите от поредицата "Основи на програмирането", нови издания, събития и инициативи:

[fb.com/IntroProgrammingBooks](https://fb.com/IntroProgrammingBooks)

## Лиценз и разпространение

Книгата се разпространява **безплатно** в електронен формат под отворен лиценз [CC-BY-NC-SA](#).

Книгата се издава и разпространява **на хартия** от СофтУни и хартиено копие може да се закупи от receptionта на СофтУни (вж. <https://softuni.bg/contacts>).

**Сурс кодът** на книгата може да се намери в GitHub хранилището на проекта: <https://github.com/SoftUni/Programming-Basics-Book-Python-BG>.

Международен стандартен номер на книга ISBN: 978-619-00-0806-4.

## Докладване на грешки

Ако откриете **грешки**, неточности или дефекти в книгата, можете да ги докладвате в официалния тракер на проекта:

<https://github.com/SoftUni/Programming-Basics-Book-Python-BG/issues>

Не обещаваме, че ще поправим всичко, което ни изпратите, но пък имаме желание **постоянно да подобряваме качеството** на настоящата книга, така че докладваните безспорни грешки и всички разумни предложения ще бъдат разгледани.

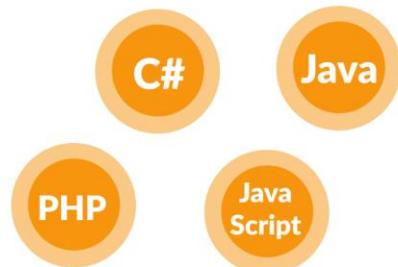
## Приятно четене!

И не забравяйте **да пишете код** в големи количества, да пробвате **примерите** от всяка тема и най-вече да решавате **задачите от упражненията**. Само с четене няма да се научите да програмирате, така че решавайте задачи здраво!

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтуни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтуни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 1. Първи стъпки в програмирането

В тази глава ще разберем **какво е програмирането** в неговата същина. Ще се запознаем с идеята за **програмни езици** и ще разгледаме **средите за разработка на софтуер** (Integrated Development Environment – накратко **IDE**) и как да работим с тях, в частност с **PyCharm**. Ще напишем и изпълним **първата си програма** на програмния език **Python**, а след това ще се упражним с няколко задачи: ще създадем конзолна програма, графично приложение и уеб приложение. Ще се научим как да проверяваме за коректност решенията на задачите от тази книга в **Judge системата на СофтУни** и накрая ще се запознаем с типичните грешки, които често се допускат при писането на код и как да се предпазим от тях.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=LUUIDcwBDss>.

## Какво означава "да програмираме"?

Да **програмираме** означава да даваме команди на компютъра какво да прави, например "да иззвири някакъв звук", "да отпечата нещо на екрана" или "да умножи две числа". Когато командите са няколко една след друга, те се наричат **компютърна програма** (или скрипт). Текстът на компютърните програми се нарича **програмен код** (или **сурс код**, или за по-кратко **код**).

## Компютърни програми

Компютърните програми представляват **поредица от команди**, които се изписват на предварително избран **език за програмиране**, например Python, C#, Java, JavaScript, Ruby, PHP, C, C++, Swift, Go или друг. За да пишем команди, трябва да знаем **синтаксиса и семантиката на езика**, с който ще работим, в нашия случай **Python**. Затова ще се запознаем със синтаксиса и семантиката на езика Python, и с програмирането като цяло в настоящата книга, изучавайки стъпка по стъпка писането на код, от по-простите към по-сложните програмни конструкции.

## Алгоритми

Компютърните програми обикновено изпълняват някакъв алгоритъм. **Алгоритмите** са последователност от стъпки, необходими за да се свърши определена работа и да се постигне някакъв очакван резултат, нещо като "рецепта". Например, ако пържим яйца, ние изпълняваме някаква рецепта (алгоритъм): загряваме мазнина в някакъв съд, чупим яйцата, изчакваме докато се изпържат, отместваме от огъня. Аналогично, в програмирането **компютърните програми изпълняват алгоритми** - поредица от команди, необходими, за да се свърши определена работа. Например, за да се подредят поредица от числа в нарастващ ред, е необходим алгоритъм, примерно да се намери най-малкото число и да се отпечата, от останалите числа да се намери отново най-малкото число и да се отпечата, и това се повтаря докато числата свършат.

За удобство при създаването на програми, за писане на програмен код (команди), за изпълнение на програмите и за други операции, свързани с програмирането, ни е необходима и среда за разработка (IDE), например PyCharm.

## Езици за програмиране, компилатори, интерпретатори и среди за разработка

Езикът за програмиране е изкуствен език (синтаксис за изразяване), предназначен за задаване на команди, които искаме компютъра да прочете, обработи и изпълни. Чрез езиците за програмиране пишем поредици от команди (**програми**), които задават какво да прави компютъра. Изпълнението на компютърните програми може да се реализира с **компилатор** или с **интерпретатор**.

**Компилаторът** превежда кода от програмен език на **машинен код**, като за всяка от конструкциите (командите) в кода избира подходящ, предварително подготвен фрагмент от машинен код и междувременно **проверява за грешки в текста на програмата**. Заедно компилираните фрагменти съставят програмата в машинен код, както я очаква микропроцесора на компютъра. След като е компилирана програмата, тя може да бъде директно изпълнена от микропроцесора в кооперация с операционната система.

При компилируемите езици за програмиране **компилирането на програмата** се извършва задължително преди нейното изпълнение и по време на компилация се откриват синтактичните грешки (грешно зададени команди). С компилатор работят езици като C++, C#, Java, Swift и Go.

Някои езици за програмиране не използват компилатор, а се **интерпретират директно** от специализиран софтуер, наречен "интерпретатор". **Интерпретаторът** е "**програма за изпълняване на програми**", написани на някакъв програмен език. Той изпълнява командите на програмата една след друга, като разбира не само от единични команди и поредици от команди, но и от другите езикови конструкции (проверки, повторения, функции и т.н.).

Езици като **Python**, **PHP** и **JavaScript** работят с интерпретатор и се изпълняват директно без да се компилират. Поради липса на предварителна компилация, при интерпретираните езици **грешките се откриват по време на изпълнение**, след като програмата започне да работи, а не предварително. При интерпретираните езици за програмиране по-бързо можем да променяме кода и да го изпълняваме отново (например при отстраняване на грешка).

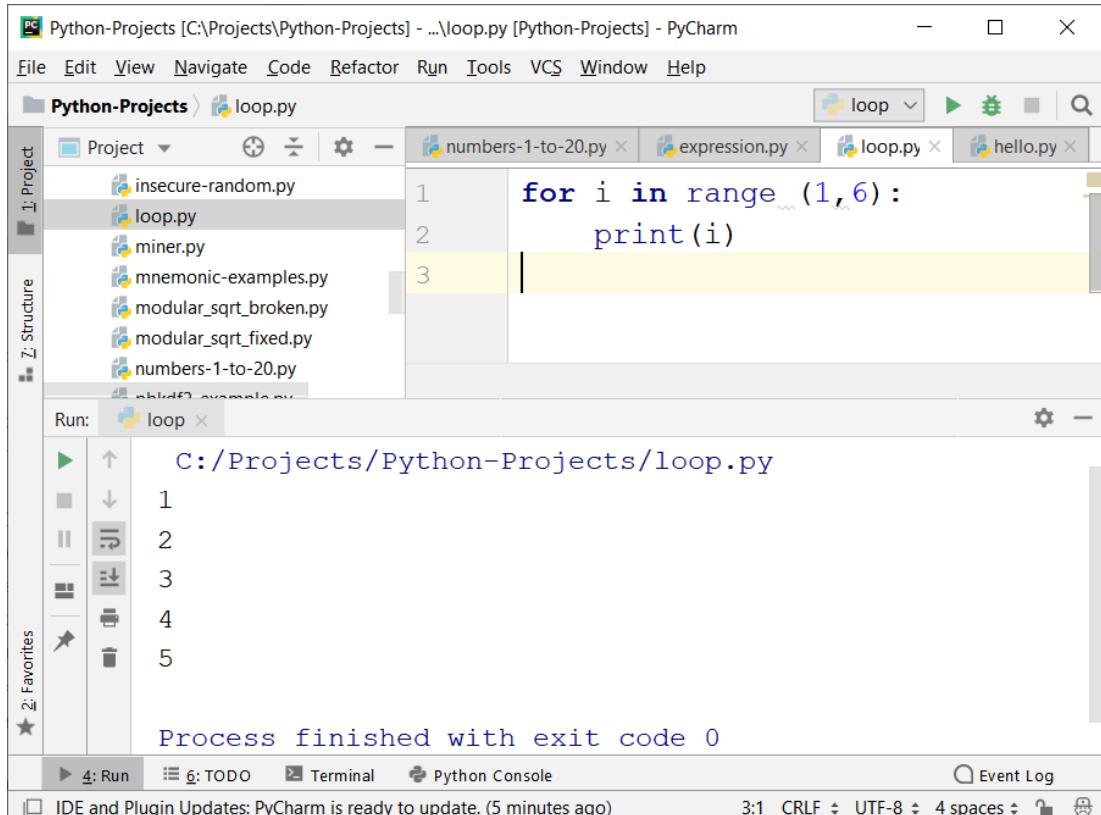
Като цяло **компилируемите езици работят по-бързо от интерпретираните**. Разликата в скоростта на изпълнение на програмите понякога е минимална, а понякога е огромна. При интерпретираните езици за програмиране **по-бързо можем да променяме кода и да го изпълняваме отново** (например за да отстраним грешка), докато компилируемите езици са по-бавни след промяна на кода заради нуждата от компилация.

**Средата за програмиране** (Integrated Development Environment – IDE, интегрирана среда за разработка) е съвкупност от традиционни инструменти за разработване

на софтуерни приложения. В средата за разработка пишем код, компилираме и изпълняваме програмите. Средите за разработка интегрират в себе си **текстов редактор** за писане на кода, **език за програмиране**, **компилатор** или **интерпретатор** и **среда за изпълнение** за изпълнение на програмите, **дебъгер** за проследяване на програмата и търсене на грешки, **инструменти за дизайн на потребителски интерфейс** и други инструменти и добавки.

**Средите за програмиране** са удобни, защото интегрират всичко необходимо за разработката на програмата, без да се напуска средата. Ако не ползваме среда за разработка, ще трябва да пишем кода в текстов редактор, да го компилираме с команда от конзолата, да го изпълняваме с друга команда от конзолата и да пишем още допълнителни команди, когато се налага, и това ще ни губи време. Затова повечето програмисти ползват IDE в ежедневната си работа.

За програмиране на **езика Python** най-често се ползва средата за разработка **PyCharm**, която се разработва и разпространява от JetBrains, и може да се изтегли от: <https://jetbrains.com/pycharm>. Ето как изглежда средата **PyCharm** в действие:



Алтернативи на PyCharm са **Visual Studio Code** (<https://code.visualstudio.com>), **Atom** (<https://atom.io>), **Eclipse** (<https://eclipse.org>) + приставката **PyDev** (<http://pydev.org>) и други. В настоящата книга ще използваме средата за разработка PyCharm.

Алтернатива на настолните среди за разработка (IDE) са **онлайн средите** за Python програмиране като [Repl.it](#) и [PythonAnywhere](#). Те вършат отлична работа за

тестване на кратки примери или когато нямаме възможност да инсталираме локално среда за разработка и Python интерпретатор или когато искаме лесно да споделим кода си с колеги. Ето как се изпълнява Python код в **Repl.it**:

```
main.py
1 import math
2 r = 5
3 area = math.pi * r * r
4 print("area = ", area)

[GCC 4.8.2] on linux
area = 78.53981633974483
```

## Езици от ниско и високо ниво, среди за изпълнение (Runtime Environments)

Програмата в своята същност е **набор от инструкции**, които карат компютъра да свърши определена задача. Те се въвеждат от програмиста и се **изпълняват безусловно от машината**.

Съществуват различни видове **езици за програмиране**. С езиците от най-ниско ниво могат да бъдат написани **самите инструкции**, които **управляват процесора**, например с езика "**assembler**". С езици от малко по-високо ниво като **C** и **C++** могат да бъдат създадени операционна система, драйвери за управление на хардуера (например драйвер за видеокарта), уеббраузъри, компилатори, двигатели за графика и игри (game engines) и други системни компоненти и програми. С езици от още по-високо ниво като **C#, Python** и **JavaScript** се създават приложни програми, например програма за четене на поща или чат програма.

**Езиците от ниско ниво** управляват директно хардуера и изискват много усилия и огромен брой команди, за да свършат единица работа. **Езиците от по-високо ниво** изискват по-малко код за единица работа, но нямат директен достъп до хардуера. На тях се разработва приложен софтуер, например уеб приложения и мобилни приложения.

Болшинството софтуер, който използваме ежедневно, като музикален плеър, видеоплеър, GPS програма и т.н., се пише на **езици за приложно програмиране**, които са от високо ниво, като Python, JavaScript, C#, Java, C++, PHP и др.

Python е интерпретиран език, а това означава, че пишем команди, които се изпълняват директно след стартиране на програмата. Това означава, че ако сме допуснали грешка при писането на код, ще разберем едва след стартиране на програмата и достигането до грешната команда. Тук на помощ идват IDE-тата, като PyCharm, които проверяват кода ни, още докато пишем и ни алармирят за евентуални проблеми. Когато сме написали кода си и искаем да го тестваме, можем да го запаметим във файл с разширение **.py**, примерно **example.py**.

## Python интерпретатор: инсталация и използване

За да програмираме на езика Python, първо трябва да си инсталираме Python интерпретатор. Той изпълнява Python командите и програмите и е абсолютно необходим, ако ще пишем на езика Python на нашия компютър.

### Инсталация на Python интерпретатора

Инсталацията на Python е много лесна. Отиваме на <https://python.org/downloads> и изтегляме последната версия за нашата платформа:



The screenshot shows the Python Software Foundation website at <https://www.python.org/downloads/>. The main header features the Python logo and the word "python™". Below the header is a navigation bar with "Search" and "GO" buttons. The main content area has a heading "Download the latest version for Windows" and a prominent yellow button labeled "Download Python 3.7.2". Below this, text links to "Windows", "Linux/UNIX", "Mac OS X", and "Other" operating systems. Further down, there are links for "Pre-releases" and "Looking for Python 2.7? See below for specific releases".

В Windows среда инсталацията е стандартната с [Next], [Next] и [Finish]. В Linux се използва пакетния инсталатор, примерно **sudo apt-get install python3**.



Използвайте **Python версия 3.x или по-висока**. Python 2 е остатяла технология и макар и да е достъпен по подразбиране в много системи, той е проблемен и много от примерите от тази книга няма да работят.

## Стартиране на Python интерпретатора

След като Python интерпретаторът е вече инсталиран, можем да го стартираме и да си поиграем с него. В Windows среда използвайте [Start] менюто и намерете току-що инсталираното приложение примерно "Python 3.7 (64-bit)". В Linux / MacOS среда напишете на конзолата команда **python3**. Ето как би могъл да изглежда Python интерпретаторът в Windows среда:

>> 2+3' is shown at the bottom."/>

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
```

## Работа с Python интерпретатора

Python е **интерпретатор** и с него може се работи в **команден режим**: пишем **команда** и той я изпълнява и връща **отговора**. Най-простата команда е да накараме Python да пресметне числен израз, примерно **2+3**:

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> 
```

Виждаме, че отговорът е правилен: **5**.

Нека пробваме още няколко команди: да запишем стойност **5** в променлива с име **a** и да отпечатаме след това стойността **2 \* a**:

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> a = 5
>>> 2 * a
10
>>> 
```

Поиграйте си сами с Python интерпретатора. Опитайте да изчислите **1234567890 \* 234567890 \* 34567890**. Успяхте ли? Опитайте с **грешна команда** (например напишете си името). Опитайте да пресметнете **2 \*\* 20** и помислете какво ли е това. Опитайте да отпечатате **текста** "Здравей питоне, как си?" с команда **print("some text")**. Получава ли се?

## Компютърни програми

Както вече споменахме, програмата е **последователност от команди**, иначе казано тя описва поредица от пресмятания, проверки, повторения и всякакви подобни операции, които целят постигане на някакъв резултат.

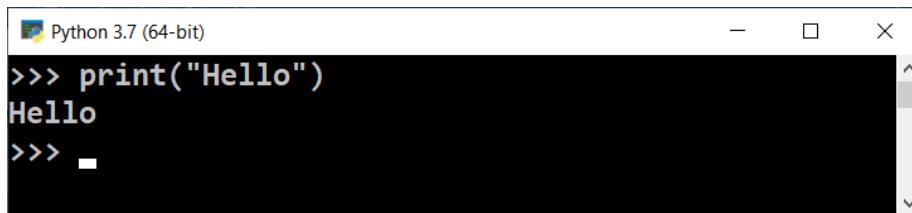
Програмата се пише в текстов формат, а самият текст на програмата се нарича **сорс код** (source code). Той се запазва във файл с разширение **.py** (например **main.py**), след което може да се изпълни през вашия браузър или през **конзолата** с помощта на **Python Shell**. При **скриптови и интерпретериеми езици**, като JavaScript, Python и PHP, сорс кода се изпълнява постъпково от интерпретатор.

### Пример: програма, която печата "Hello"

Да започнем с много прост пример за кратка Python програма. Нашата първа програма ще отпечата думата "**Hello**" ето така:

```
print("Hello")
```

Можем да я изпълним като я напишем в Python интерпретатора:



The screenshot shows a terminal window titled "Python 3.7 (64-bit)". The command line displays ">>> print('Hello')". The output window shows the word "Hello" printed. The window has standard minimize, maximize, and close buttons at the top right.

### Пример: програма, която свири музикалната нота "ла"

Нашата следваща програма ще се състои от единична Python команда, която свири музикалната нота "ла" (432 херца) с продължителност половин секунда (500 милисекунди):

```
import winsound  
  
winsound.Beep(432, 500)
```

В Windows среда **ще чуем звук**. Уверете се, че звукът на компютъра ви е пуснат. В Linux и macOS примерът няма да проработи.

### Пример: програма, която свири поредица от музикални ноти

Можем да усложним предходната програма, като зададем за изпълнение повтарящи се в цикъл команди за свирене на поредица от ноти с нарастваща височина:

```
import winsound
```

```
for i in range(200, 4000, 200):
    winsound.Beep(i, 300)
```

В горния пример караме компютъра да свири една след друга за много кратко (по 300 милисекунди) всички ноти с височина 200, 400, 600 и т.н. херца до достигане на 4000 херца. Резултатът от програмата е свирене на нещо като мелодия.

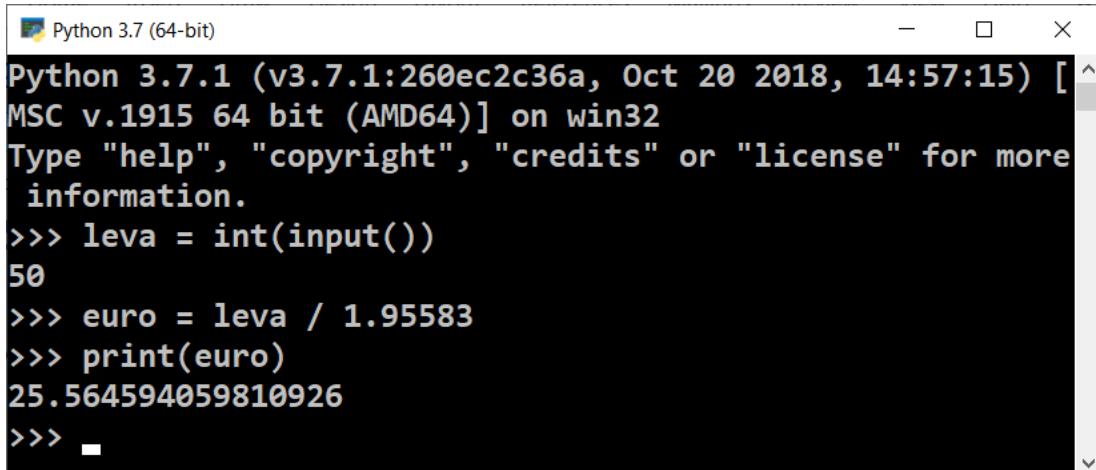
Как работят повторенията (циклите) в програмирането ще научим в главата "[Цикли](#)", но засега приемете, че просто повтаряме някаква команда много пъти.

## Пример: програма, която конвертира от левове в евро

Да разгледаме още една проста програма, която прочита от потребителя някаква сума в лева (цяло число), конвертира я в евро (като я разделя на курса на еврото) и отпечатва получения резултат. Това е програма от **три поредни команди**. Въведете ги и ги изпълнете **една след друга**:

```
leva = int(input())
euro = leva / 1.95583
print(euro)
```

Ето как би могъл да изглежда **резултатът** от горната серия команди:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> leva = int(input())
50
>>> euro = leva / 1.95583
>>> print(euro)
25.564594059810926
>>> -
```

Разгледахме **три примера за компютърни програми**: единична команда, серия команди в цикъл и поредица от три команди. Нека сега преминем към по-интересното: как можем да пишем собствени програми на **Python** и как можем да ги изпълняваме?

## Как да напишем конзолна програма?

Нека преминем през **стъпките за създаване и изпълнение на компютърна програма**, която чете и пише своите данни от и на текстова конзола (прозорец за въвеждане и извеждане на текст). Такива програми се наричат "**конзолни**". Преди

това, обаче, трябва първо да си **инсталираме и подгответим средата за разработка**, в която ще пишем и изпълняваме Python програмите от тази книга и упражненията към нея.

## Среда за разработка (IDE)

Както вече стана дума, за да програмираме ни е нужна **среда за разработка** - **Integrated Development Environment** (IDE). Това всъщност е редактор за програми, в който пишем програмния код и можем да го изпълняваме, да виждаме грешките, да ги поправяме и да стартираме програмата отново.

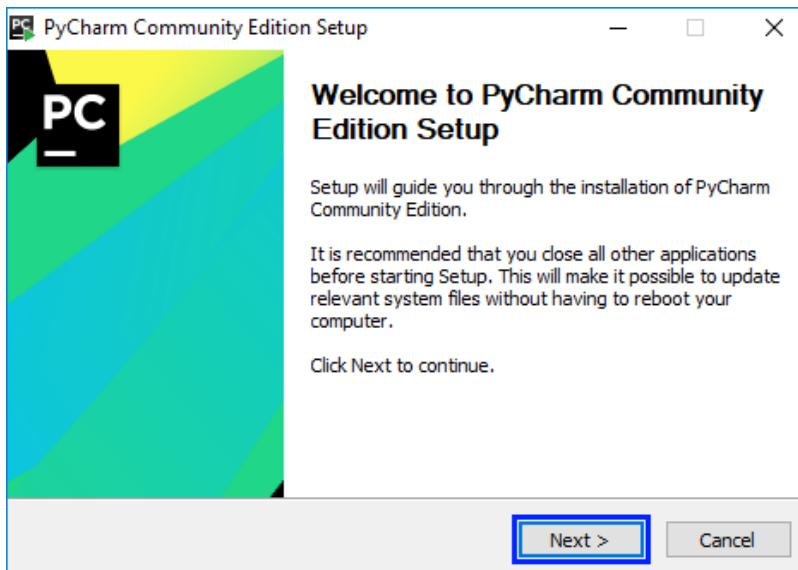
- За програмиране на Python използваме средата **PyCharm** за операционните системи Windows, Linux или macOS.
- Ако програмираме на Java, подходящи са средите **IntelliJ IDEA**, **Eclipse** или **NetBeans**.
- Ако ще пишем на C#, можем да използваме средата **Visual Studio**.

## Инсталация на PyCharm Community

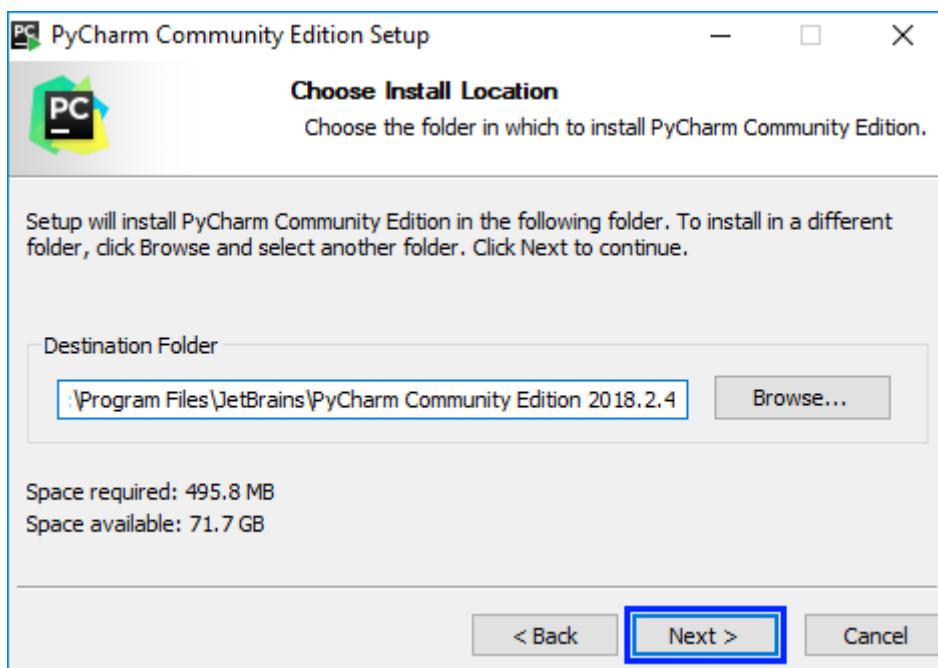
Започваме с инсталацията на интегрираната среда **PyCharm Community** (в нашия пример ще използваме версия 2018, актуална към октомври 2018 г.).

**Community** версията на PyCharm се разпространява безплатно от JetBrains и може да бъде изтеглена от: <https://www.jetbrains.com/pycharm/download>. Инсталацията е типичната за Windows с [Next], [Next] и [Finish]. Има и Linux / macOS версия.

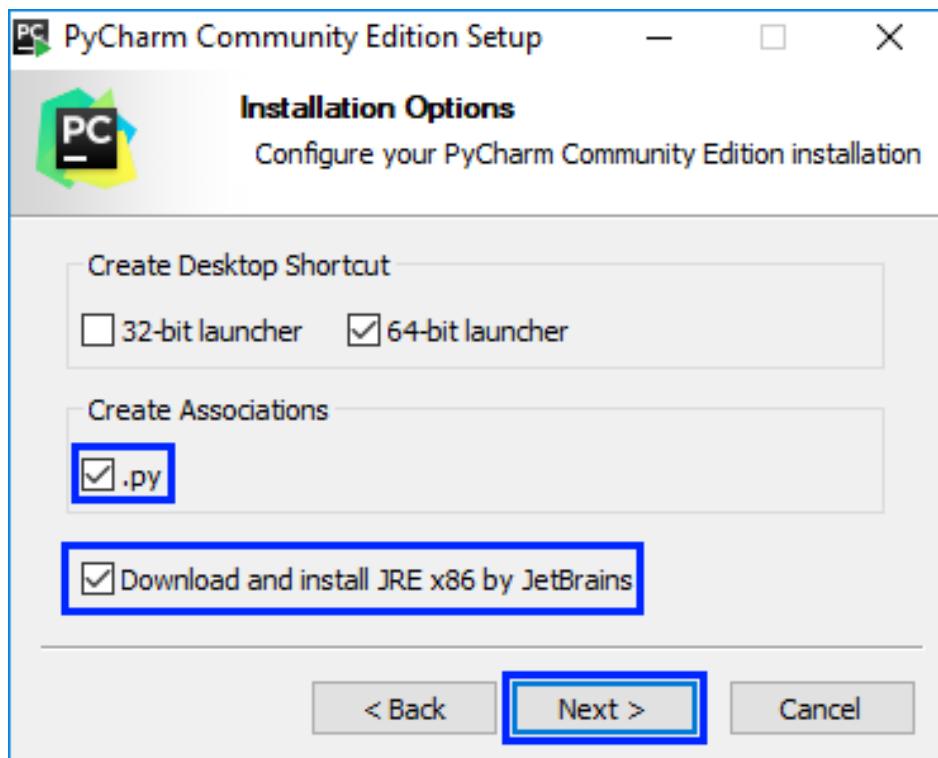
В следващите редове подробно са описани **стъпките за инсталация на PyCharm** (версия Community 2018). След като свалим инсталационния файл и го стартираме, се появява следния екран:



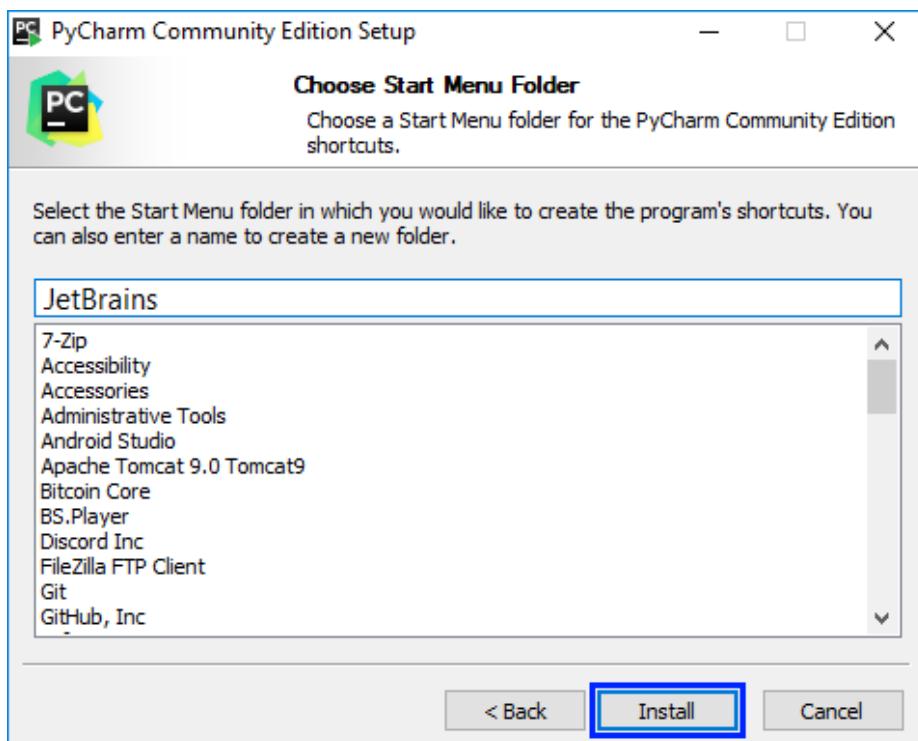
Натискаме бутона **[Next]**, след което ще видим прозореца долу:



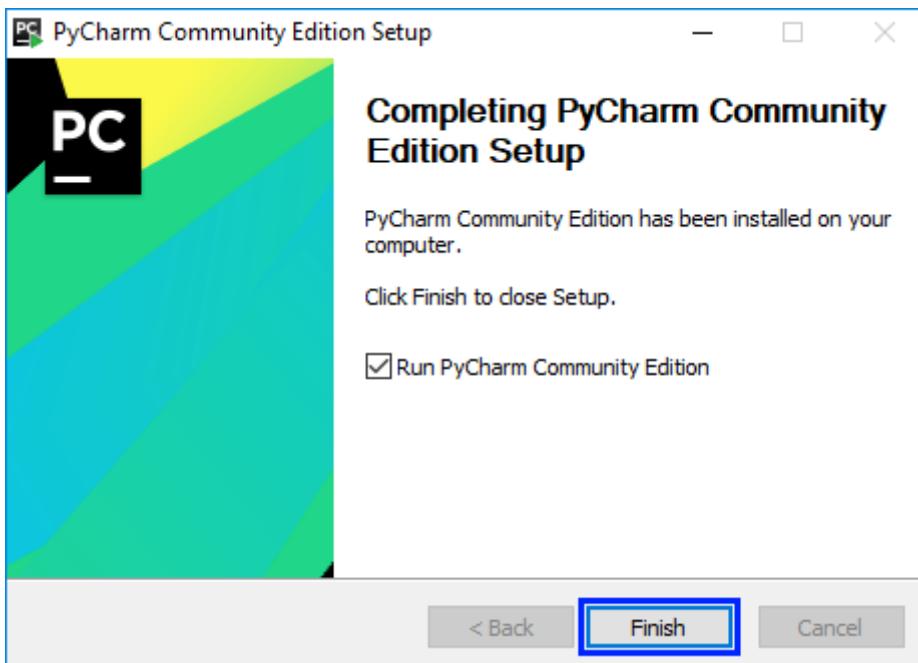
Зарежда се прозорец с инсталационния панел на PyCharm:



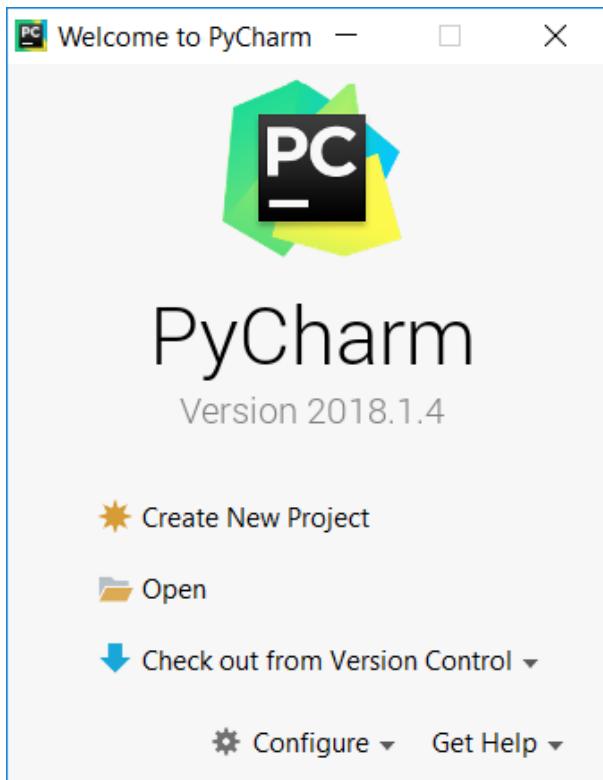
Слагаме отметка на прям път в зависимост от нашата операционна система, [.py] и [Download and install JRE x86 by JetBrains], след което натискаме бутона [Next]:



Натискаме бутона [Install].



След като PyCharm се инсталира, ще се появи информативен екран, където можем да сложим отметка ([Run PyCharm ...]), за да го стартираме. След **старта на PyCharm** излиза екран като този по-долу:



Това е всичко. Готови сме за работа с PyCharm.

## По-стари версии на PyCharm

Можем да използваме и по-стари версии на PyCharm (например версия 2016 или 2012), но **не е препоръчително**, тъй като в тях не се съдържат някои от по-новите възможности за разработка и не всички примери от книгата ще тръгнат.

## Онлайн среди за разработка

Съществуват и **алтернативни среди за разработка онлайн**, директно в нашия уеб браузър. Тези среди не са много удобни, но ако нямаете друга възможност, може да стартирате обучението си с тях и да си инсталирате PyCharm по-късно. Ето някои линкове:

- За езика Python сайтът **Tutorials Point** позволява писане на код и изпълнението му онлайн: [https://tutorialspoint.com/online\\_python\\_ide.php](https://tutorialspoint.com/online_python_ide.php).
- Друга добра онлайн среда за писане и изпълнение на Python код е Repl.it: <https://repl.it/languages/python3>.
- За Java можем да използваме следното онлайн Java IDE: <https://www.compilejava.net>.
- За JavaScript можем да пишем JS код директно в конзолата на даден браузър с натискане на [F12].

## Проекти в PyCharm

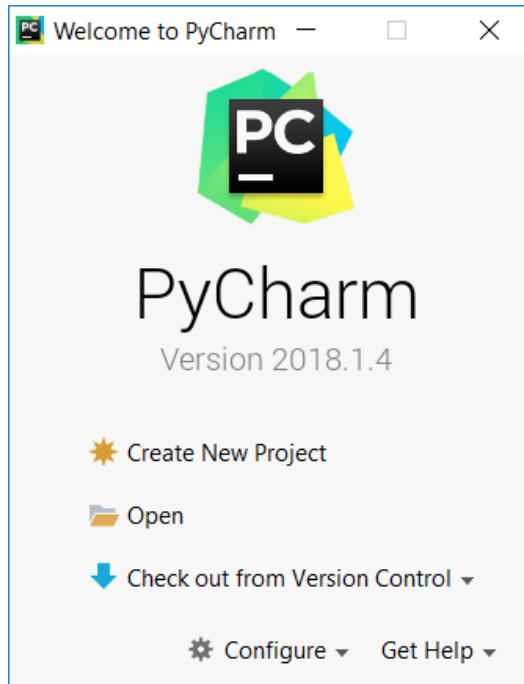
Преди да започнем да работим с PyCharm, нека се запознаем с понятието **PyCharm Project**.

**PyCharm Project** представлява "проекта", върху който работим. В началото това ще са нашите конзолни програми, които ще се научим да пишем с помощта на настоящата книга, ресурсите към нея и в курса Programming Basics в SoftUni. При по-задълбочено изучаване и с времето и практиката, тези проекти ще преминат в приложения, уеб приложения и други разработки. Проектът в PyCharm **логически групира множество файлове**, изграждащи дадено приложение или компонент.

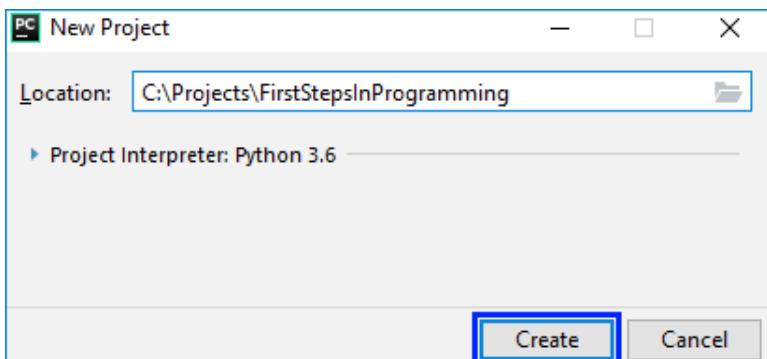
Един Python проект съдържа един или няколко Python сорс файла (**.py**), конфигурационни файлове и други ресурси. Във всеки Python сорс файл може да има един или повече **класове или функции**. В **класовете** има **функции** (действия), а те се състоят от **поредици от команди**. Изглежда сложно, но при големи проекти такава структура е много удобна и позволява добра организация на работните файлове.

## Пример: създаване на конзолна програма "Hello Python"

Да се върнем на нашата конзолна програма. Вече имаме PyCharm и можем да го стартираме. След това създаваме нов проект: [[Create New Project](#)].



Задаваме **смислено име** на проекта, който ще съдържа нашата програма, например **FirstStepsInProgramming**:



PyCharm ще създаде за нас **празен проект**, в който може да добавяме Python файлове. Добавяме нов Python файл ([File] или десен бутон на проекта ни → [New] → [Python File]) и му задаваме сmisлено име, например **HelloPython.py**.

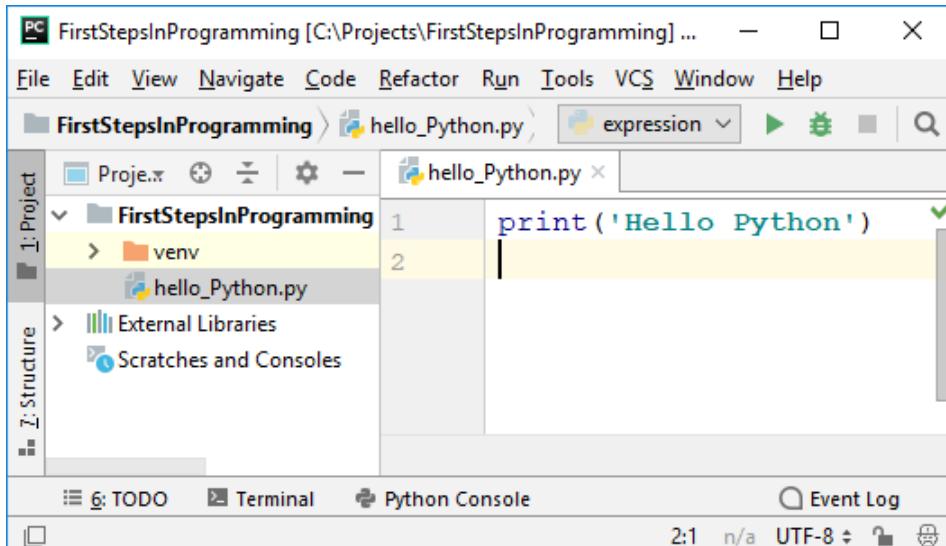
Можем да настроим версията и местоположението на **Python интерпретатора**, който ще бъде използван. Задължително е да имаме Python интерпретатор, ако ще изпълняваме написаните Python програми. Можем да имаме повече от един Python версии на един и същ компютър. За леснота, в началото, най-добре е да имате **само един Python**, последна версия, и да ползвате само него навсякъде. Ако нямате инсталиран Python, инсталрайте го преди да продължите.

## Писане на програмен код

Писането на Python код не изисква никаква допълнителна подготовка от това, което вече направихме - да си създадем файл с разширение **.py**. Затова директно пристъпваме към изписването на първия ред код. Изписваме следната команда:

```
print('Hello Python')
```

Ето как би могла да изглежда нашата програма в PyCharm:



Командата `print('Hello Python')` на езика Python означава да изпълним отпечатване (`print(...)`) върху конзолата и да отпечатаме текстово съобщение **Hello Python**, което трябва да оградим с кавички, за да поясним, че това е текст.

Тази команда е много типична за програмирането: извикваме функцията `print(...)` и ѝ подаваме като параметър текстов липерал '**Hello Python**'.

## Стартиране на програмата

За стартиране на програмата натискаме [**Ctrl + Shift + F10**] или десен бутон на мишката и [**Run**]. Ако няма грешки, програмата ще се изпълни. Резултатът ще се изпише на конзолата:

The screenshot shows the PyCharm 'Run' tool window. The title bar says 'Run: hello\_Python'. Below it, there are several icons: play (green triangle), up arrow, down arrow, pause (double vertical bars), and stop (red circle). The main pane displays the command line output:  
 C:\Users\MirelaDamyanova\AppData\Local\Programs\Python  
 Hello Python  
 Process finished with exit code 0

Въсъщност, изхода от програмата е следното текстово съобщение:

```
Hello Python
```

Съобщението "Process finished with exit code 0" се изписва допълнително на най-долния ред на конзолата на PyCharm след като програмата завърши, за да ни покаже, че програмата се е изпълнила без грешки.

## Тестване на програмата в Judge системата

Тестването на задачите от тази книга е автоматизирано и се осъществява през Интернет, от сайта на Judge системата: <https://judge.softuni.bg>. Оценяването на задачите се извършва на момента от системата. Всяка задача минава поредица от тестове, като всеки успешно преминат тест дава предвидените за него точки. Тестовете, които се подават на задачите, са скрити.

The screenshot shows the Judge system interface for the '01. Hello Python' task. On the left, there's a code editor with two lines of code: '1 print('Hello Python')' and '2'. The first line is highlighted with a blue border. On the right, there are performance limits: 'Allowed working time: 0.100 sec.', 'Allowed memory: 16.00 MB', 'Size limit: 16.00 KB', and 'Checker: Trim ?'. At the bottom right are two buttons: 'Python code' with a dropdown arrow and 'Submit'.

Горната програма може да тестваме тук: <https://judge.softuni.bg/Contests/Practice/Index/1046#0>. За целта поставяме целия сурс код на програмата в черното поле и избираме **Python code**, както е показано по-горе.

Изпращаме решението за оценяване с бутона [**Изпрати**] (или [**Submit**]). Системата връща резултат след няколко секунди в таблицата с изпратени решения. При необходимост може да натиснем бутона за обновяване на резултатите [**Refresh**], който се намира в горната дясна част на таблицата с изпратени за проверка решения:

Submissions		
<b>1</b>	<b>Refresh</b>	
Points	Time and memory used	Submission date
100 / 100	Memory: 8.20 MB Time: 0.056 s	20:49:18 30.09.2018
0 / 100	Memory: 8.20 MB Time: 0.071 s	20:48:40 30.09.2018

В таблицата с изпратените решения Judge системата ще покаже един от следните **възможни резултати**:

- **Брой точки** (между 0 и 100), когато предаденият код се компилира успешно (няма синтактични грешки) и може да бъде тестван.
  - При **вярно решение** всички тестове са маркирани в зелено и получаваме **100 точки**.
  - При **грешно решение** някои от тестовете са маркирани в червено и получаваме непълен брой точки или 0 точки.
- При грешна програма ще получим **съобщение за грешка** по време на компилация.

## Как да се регистрирам в SoftUni Judge?

Използваме идентификацията си (Username + Password) от сайта [softuni.bg](https://softuni.bg). Ако нямate СофтУни регистрация, направете си. Отнема само минутка - стандартна регистрация в Интернет сайт.

## Тествайте програмите за свирене на ноти

Сега, след като вече знаете как да изпълнявате програми, можете да тествате примерните програми по-горе, които свирят музикални ноти. Позабавлявайте се, пробвайте тези програми. Пробвайте да ги промените и да си поиграете с тях.

Заменете командата `print('Hello Python')` с команда `winsound.Beep(432, 500)`, като най-отгоре добавите `import winsound`, и стартирайте програмата. Проверете дали ви е включен звука на компютъра и дали е усилен. Ако работите в онлайн среда за разработка, няма да чуете звук, защото програмата не се изпълнява на вашия компютър, а някъде другаде. Пакетът `winsound` може да не работи под някои операционни системи като Linux и macOS.

## Типични грешки в Python програмите

Една от често срещаните грешки при начинаещите е бъркането на **главни и малки букви**, а те имат значение при извикване на командите и тяхното правилно функциониране. Ето пример за такава грешка:

```
Print('Hello Python')
```

В горния пример **Print** е изписано грешно и трябва да се поправи на **print**.

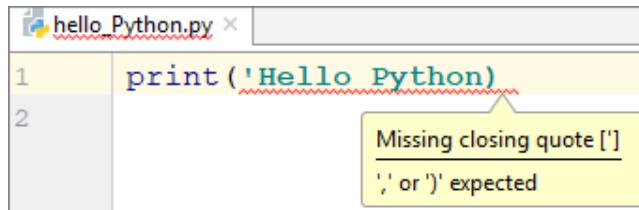


В езика Python не се слага **точка и запетая** (`:`) в края на командите. Командите се разделят една от друга чрез **нов ред или индентация** (отстъп).

Липсваща **кавичка** или липса на **отваряща** или **затваряща скоба** също може да се окажат проблеми. Проблемът води до **неправилно функциониране** на програмата или въобще до нейното неизпълнение. Този пропуск трудно се забелязва при по-обемен код. Ето пример за грешна програма:

```
print('Hello Python)
```

Тази програма ще даде **грешка** и кода ще бъде подчертан, за да се насочи вниманието на програмиста към грешката, която е допуснал (пропуснатата затваряща кавичка):



## Какво научихме от тази глава?

На първо място научихме какво е програмирането - задаване на команди, изписани на **компютрен език**, които машината разбира и може да изпълни. Разбрахме още какво е **компютърната програма** - тя представлява **поредица** от **команди**, подредени една след друга. Запознахме се с **езика за програмиране Python** на базисно ниво и как да създаваме прости конзолни **програми** с PyCharm. Проследихме и **структурата на програмния код** в езика Python. Видяхме как да

печатаме с функцията **print(...)** и как да стартираме програмата си с [Ctrl + Shift + F10]. Научихме се да тестваме кода си в **SoftUni Judge** системата.

Добра работа! Да се захващаме с **упражненията**. Нали не сте забравили, че програмиране се учи с много писане на код и решаване на задачи? Да решим няколко задачи, за да затвърдим наученото.

## Упражнения: първи стъпки в програмирането

Добре дошли в упражненията. Сега ще напишем няколко конзолни програми, с които ще направим още няколко първи стъпки в програмирането, след което ще покажем как можем да програмираме нещо по-сложно - програми с графичен и уеб потребителски интерфейс.

### Задача: конзолна програма “Expression”

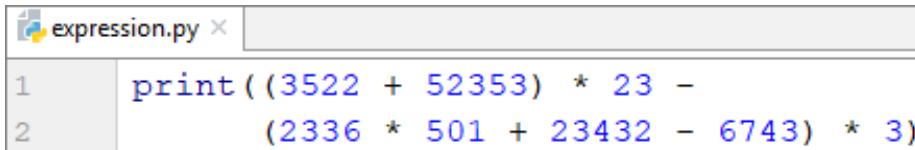
Да се напише конзолна **Python** програма, която **пресмята** и отпечатва стойността на следния числен израз:

$$(3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3$$

Забележка: не е разрешено да се пресметне стойността предварително (например с Windows Calculator).

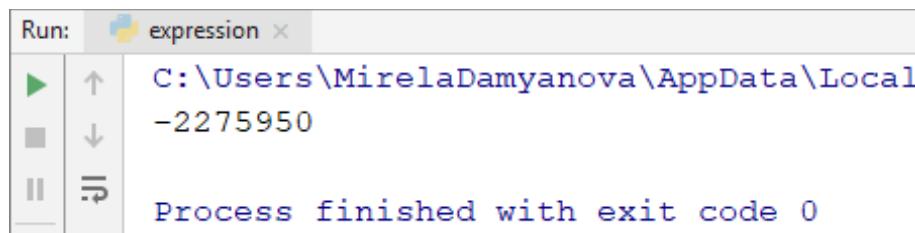
### Насоки и подсказки

Създаваме нов Python файл с име **expression**. След това трябва да **напишем** кода, който да изчисли горния числен израз и да отпечата на конзолата стойността му. Подаваме горния числен израз в скобите на команда **print(...)**:



```
expression.py
1 print((3522 + 52353) * 23 -
2           (2336 * 501 + 23432 - 6743) * 3)
```

Стартираме програмата с [Ctrl + Shift + F10] и проверяваме дали резултата е същия като на картинката:



```
Run: expression
C:\Users\MirelaDamyanova\AppData\Local
-2275950
Process finished with exit code 0
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1046#1>.

## 02. Expression

```
1 print((3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3)
2
```

Allowed working time: 0.100 sec.  
 Allowed memory: 16.00 MB  
 Size limit: 16.00 KB  
 Checker: Numbers Checker 

Python code

[Submit](#)

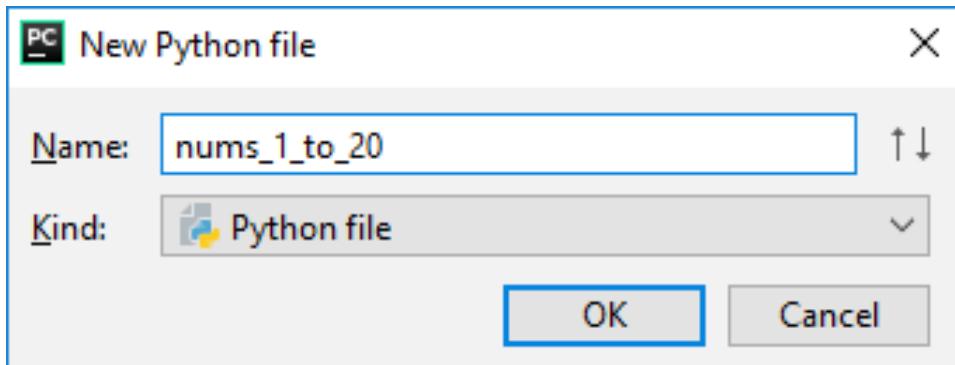
Submissions		
  <b>1</b>  		
Points	Time and memory used	Submission date
 100 / 100	Memory: 8.20 MB Time: 0.056 s	21:06:39 30.09.2018 
   		

### Задача: числата от 1 до 20

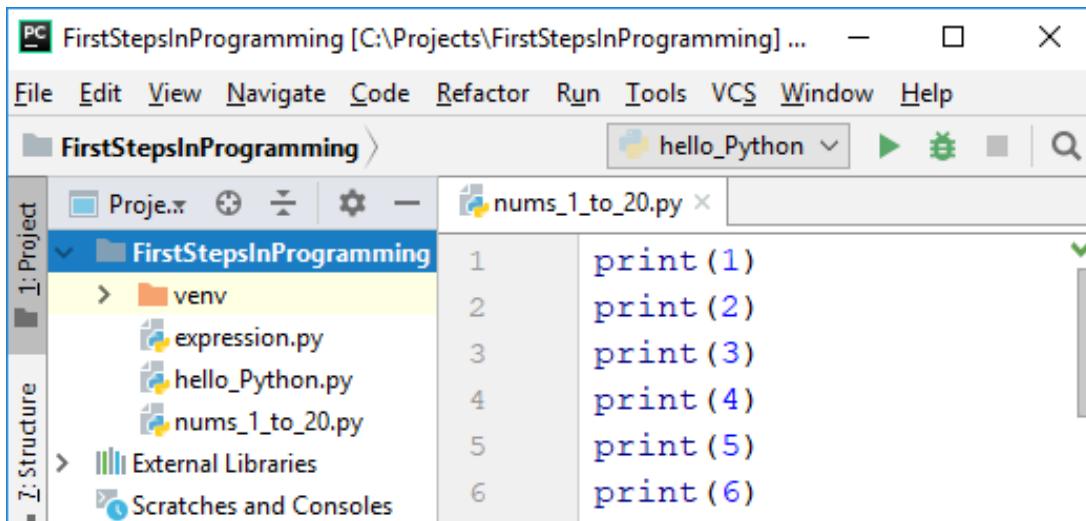
Да се напише Python конзолна програма, която отпечатва числата от 1 до 20 на отделни редове на конзолата.

#### Насоки и подсказки

Създаваме Python файл с име `nums_1_to_20.py`:



Във файла изписваме 20 команди `print(...)`, всяка на отделен ред, за да отпечататем числата от 1 до 20 едно след друго. По-досетливите от вас, сигурно се питат дали няма по-умен начин. Спокойно, има, но за него по-късно.



Сега **стартираме програмата** и поверьваме дали резултатът е какъвто се очаква да бъде:

```

1
2
...
20

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1046#2>.

Сега помислете дали може да напишем програмата по **по-умен начин**, така че да не повтаряме 20 пъти една и съща команда. Потърсете в Интернет информация за "[for loop Python](#)".

## Задача: триъгълник от 55 звездички

Да се напише Python конзолна програма, която отпечатва триъгълник от 55 звездички, разположени на 10 реда:

```

*
**
***
****
*****
******
*****
*****
*****
*****
```

## Насоки и подсказки

Създаваме нов Python файл с име `triangle_of_55_stars.py`. В него трябва да напишем код, който печата триъгълника от звездички, например чрез 10 команди, като посочените по-долу:

```
print('*')
print('**')
...
...
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1046#3>.

Опитайте да подобрите решението, така че да няма много повтарящи се команди. Може ли това да стане с **for** цикъл? Успяхте ли да намерите умно решение (например с цикъл) на предната задача? При тази задача може да се ползва нещо подобно, но малко по-сложно (два цикъла един в друг). Ако не успеете, няма проблем, ще учим цикли след няколко глави и ще си спомните за тази задача тогава.

## Задача: лице на правоъгълник

Да се напише Python програма, която получава две числа `a` и `b`, пресмята и отпечатва лицето на правоъгълник със страни `a` и `b`.

### Примерен вход и изход

a	b	area
2	7	14

a	b	area
12	5	60

a	b	area
7	8	56

## Насоки и подсказки

Създаваме нов Python файл. За да прочетем двете числа, използваме следните две команди:



```
rectangle_area.py
1 a = int(input())
2 b = int(input())
3
4 # TODO: Пресметнете лицето и го принтирайте
```

Остава да се допише програмата по-горе, за да пресмята лицето на правоъгълника и да го отпечатва. Използвайте познатата ни вече команда `print(...)` и ѝ подайте в скобите произведението на числата `a` и `b`. В програмирането умножението се извършва с оператора `*`.

## Тествайте решението си

Тествайте решението си с няколко примера. Трябва да получите резултат, подобен на този (въвеждаме 2 и 7 като вход и програмата отпечатва като резултат 14 - тяхното произведение):

```
2
7
14
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1046#4>.

### \* Задача: квадрат от звездички

Да се напише **Python** конзолна програма, която прочита от конзолата **цяло положително число N** и отпечатва конзолата **квадрат от N звездички**, като в примерите по-долу.

#### Примерен вход и изход

Вход	Изход
3	*** * * ***

Вход	Изход
4	**** * * * * ****

Вход	Изход
5	***** * * * * * * *****

#### Насоки и подсказки

Създаваме нов **Python** файл. За да прочетем числото **N** ( $2 \leq N \leq 100$ ), използваме следния код:

```
square_of_stars.py
1 n = int(input())
2
3 # TODO: Принтирайте правоъгълника
```

Да се допише програмата по-горе, за да отпечатва квадрат, съставен от звездички. Може да се наложи да се използват **for** цикли. Потърсете информация в Интернет.

**Внимание:** тази задача е по-трудна от останалите и нарочно е дадена сега и е обозначена със звездичка, за да ви провокира да потърсите информация в Интернет. Това е едно от най-важните умения, което трябва да развивате докато учите програмирането: **да търсите информация в Интернет**. Това ще правите

всеки ден, ако работите като програмисти, така че не се плашете, а се опитайте. Ако имате трудности, можете да потърсите помощ и в СофтУни форума: <https://softuni.bg/forum>.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1046#5>.

## Конзолни, графични и уеб приложения

При конзолните приложения, както и сами можете да се досетите, **всички операции** за четене на вход и печатане на изход се **извършват през конзолата**. Там се **въвеждат входните данни**, които се прочитат от приложението, там се **отпечатват и изходните данни** след или по време на изпълнение на програмата.

Докато конзолните приложения **ползват текстовата конзола**, уеб приложениета използват **уеб-базиран потребителски интерфейс**. За да се **постигне тяхното изпълнение**, са необходими две неща - **уеб сървър** и **уеб браузър**, като **браузърът играе главната роля по визуализация на данните и взаимодействието с потребителя**. Уеб приложениета са много по-приятни за потребителя, изглеждат визуално много по-добре, използват се мишка и докосване с пръст (при таблети и телефони), но зад всичко това стои програмирането. И затова **трябва да се научим да програмираме**, и вече направихме първите си съвсем малки стъпки.

Графичните (GUI) приложения имат **визуален потребителски интерфейс**, директно върху нашия компютър или мобилно устройство, без да е необходим уеб браузър. Графичните приложения (настолни приложения или, иначе казано, desktop apps) се състоят от **един или повече графични прозореца**, в които се намират определени **контроли** (текстови полета, бутони, картички, таблици и други), **служещи за диалог** с потребителя по по-интуитивен начин. Подобни са и мобилните приложения в нашия телефон и таблет: ползваме форми, текстови полета, бутони и други контроли и ги управляване чрез програмен код. Нали затова се учим сега да пишем код: **кодът е навсякъде в разработката на софтуер**.

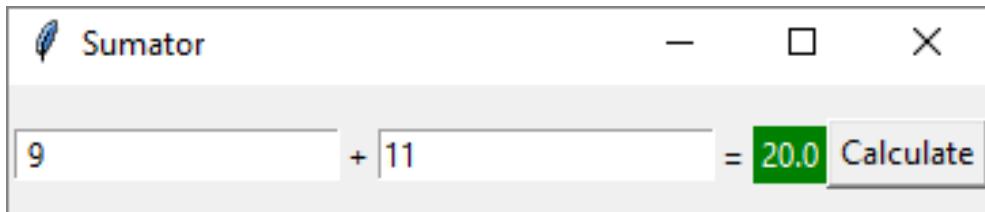
## Упражнения: графични и уеб приложения

Сега предстои да направим едно **графично приложение** (GUI application) и едно **уеб приложение** (Web application), за да надникнем в това, какво ще можем да създаваме някой ден като напреднем с програмирането и разработката на софтуер. Няма да разглеждаме детайлите по използваните техники и конструкции из основи, а само ще хвърлим поглед върху подредбата и функционалността на създаденото от нас. След като напреднем със знанията си, ще бъдем способни да правим големи и сложни софтуерни приложения и системи. Надяваме се примерите по-долу да ви запалят интереса, а не да ви откажат.

### Графично приложение: “Суматор за числа”

Да се напише **графично (GUI) приложение**, което **изчислява сумата на две числа**. След въвеждане на числата в първите две текстови полета и натискане на бутона

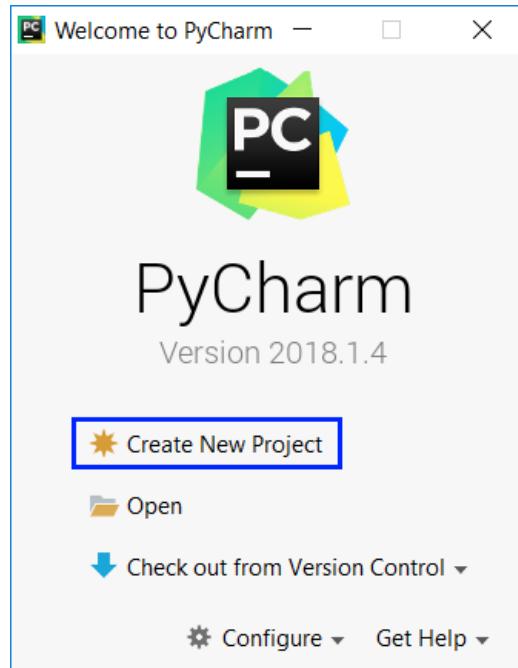
[Calculate] се изчислява тяхната сума и резултатът се показва в третото текстово поле в (зелен цвят):



За нашето приложение ще използваме стандартната Python графична библиотека TkInter (<https://wiki.python.org/moin/TkInter>), която позволява създаване на графични (GUI) приложения с езика за програмиране Python.

## Празен Python проект

В PyCharm създаваме нов Python проект с име "Summator-GUI":



Добавяме в проекта нов Python файл с име `sumator.py`. Добавяме и `tkinter`:

```
sumator.py x
1 import tkinter as tk
```

```
import tkinter as tk
```

## Създаване основата на GUI проекта

Следва да напишем кода на нашето графично приложение. Започваме с основата:

```
class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        self.pack()

    # create the application
app = Application()
app.master.title("Sumator")
app.master.minsize(width=100, height=50)

# start the program
app.mainloop()
```

Ще са ни нужни следните компоненти (контроли):

- **Entry** – ще въвежда числата за пресмятане.
- **Label** – ще ни служи за статично изобразяване на текст и резултата от пресмятането.
- **Button** – ще изчислява сумата на числата.

```
def __init__(self, master=None):
    super().__init__(master)

    self.pack()
    self.create_widgets()
```

Инициализираме и визуализираме нашите контроли (widgets):

```
def create_widgets(self):
    # create widgets
    self.firstNumberEntry = tk.Entry()
    self.plusSign = tk.Label(text="+")
    self.secondNumberEntry = tk.Entry()
    self.equalSign = tk.Label(text="=")
    self.resultLabel = tk.Label(text="Result...", bg="green", fg="white")
    self.calculateButton = tk.Button(text="Calculate")

    # place widgets
    self.firstNumberEntry.pack(side="left")
    self.plusSign.pack(side="left")
    self.secondNumberEntry.pack(side="left")
    self.equalSign.pack(side="left")
    self.resultLabel.pack(side="left")
    self.calculateButton.pack(side="left")
```

Опитваме да пуснем приложението с [Ctrl + Shift + F10] или с десен бутон на мишката + [Run]. То би трябвало да стартира, но да не функционира напълно, защото не сме написали какво се случва при натискане на бутона:



## Логика на приложението

Сега е време да напишем кода, който сумира числата от първите две полета и показва резултата в третото поле. За целта при инициализирането на бутона, добавяме команда **calculate**:

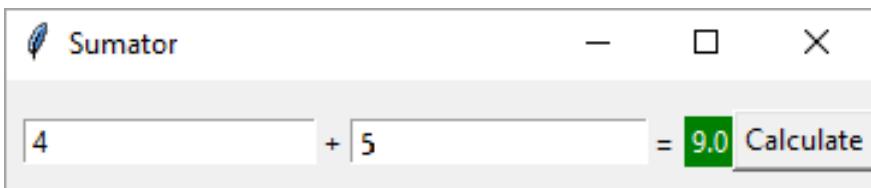
```
tk.Button(text="Calculate", command=self.calculate)
```

Написваме кода на функцията **calculate**:

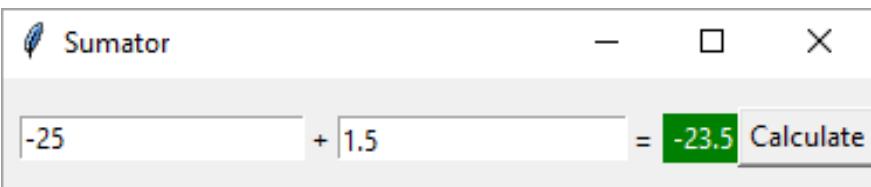
```
def calculate(self):
    first_value = float(self.firstNumberEntry.get())
    second_value = float(self.secondNumberEntry.get())
    result = first_value + second_value
    self.resultLabel.config(text=str(result),
                           bg="green", fg="white")
```

Този код взима първото число от полето **firstNumberEntry** и го запазва в променливата **first\_value**, запазва второто число от полето **secondNumberEntry** в променливата **second\_value**, след това сумира **first\_value** и **second\_value** в променливата **result** и накрая извежда текстовата стойност на променливата **result** в полето **resultLabel**.

Стартираме отново програмата с [Ctrl + Shift + F10] или с десен бутон + [Run], и проверяваме дали работи коректно. Правим опит да сметнем  $4 + 5$ . Изглежда работи:

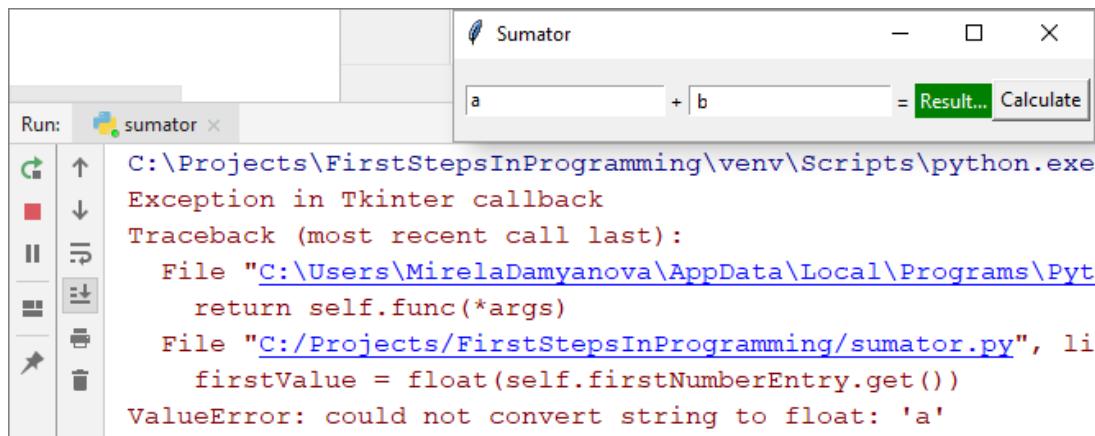


След това пресмятаме след това  $-25 + 1.5$ . Ето го и резултатът:



## Обработка на невалидни числа

Пробваме и с **невалидни числа**, напр. "a" и "b". Изглежда има проблем:

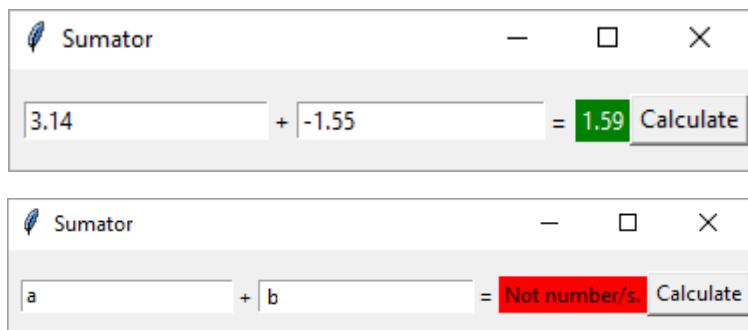


Проблемът идва от прехвърлянето на текстово поле в число. Ако стойността в полето не е число, програмата дава грешка. Можем да поправим кода, за да коригираме този проблем:

```
def calculate(self):
    try:
        first_value = float(self.firstNumberEntry.get())
        second_value = float(self.secondNumberEntry.get())
        result = first_value + second_value
        self.resultLabel.config(text=str(result),
                               bg="green", fg="white")
    except ValueError:
        self.resultLabel.config(text="Not number/s.",
                               bg="red", fg="black")
```

Горният код прихваща грешките при работа с числа (хваща изключенията) и в случай на грешка извежда стойност **Not number/s.** в полето с резултата. Стартураме отново програмата с [Ctrl + Shift + F10] или с десен бутон - [Run], и я пробваме дали работи.

Този път при грешно число резултатът е **Not number/s.** и програмата не се чупи:

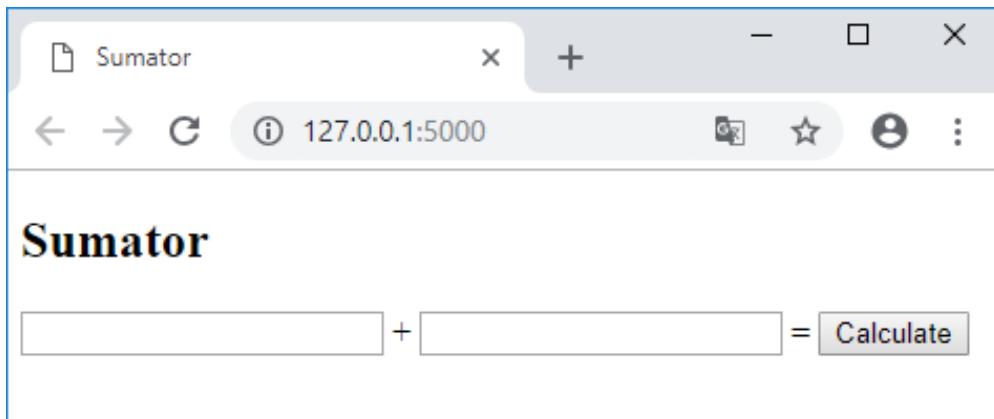


Сложно ли е? Нормално е да е сложно, разбира се. Тъкмо започваме да навлизаме в програмирането. Примерът по-горе изиска още много знания и умения, които ще развиваме в тази книга и даже и след нея. Просто си позволете да се позабавлявате с desktop програмирането. Ако не тръгва нещо, винаги може да питате във **форума на СофтУни**: <https://softuni.bg/forum>. Или продължете смело напред към следващия пример или към следващата глава от книгата. Ще дойде време, когато ще ви е лесно, но наистина трябва да вложите **усърдие и постоянно**. Програмирането се учи бавно и с много, много практика.

## Уеб приложение: “Суматор за числа”

Сега ще напищем нещо още по-сложно, но и по-интересно: уеб приложение, което **изчислява сумата на две числа**. При **въвеждане на две числа** в първите две текстови полета и натискане на бутона **[Calculate]** се **изчислява тяхната сума** и резултата се показва в третото текстово поле.

Обърнете внимание, че ще създадем **уеб-базирано приложение**. Това е приложение, което е достъпно през уеб браузър, точно както любимата ви уеб поща или новинарски сайт. Уеб приложението ще има сървърна част (back-end), която е написана на езика Python с технологията **Flask** (<http://flask.pocoo.org>) и клиентска част (front-end), която е написана на езика **HTML** (това е език за визуализация на информация в уеб браузър). Уеб приложението се очаква да изглежда приблизително по следния начин:

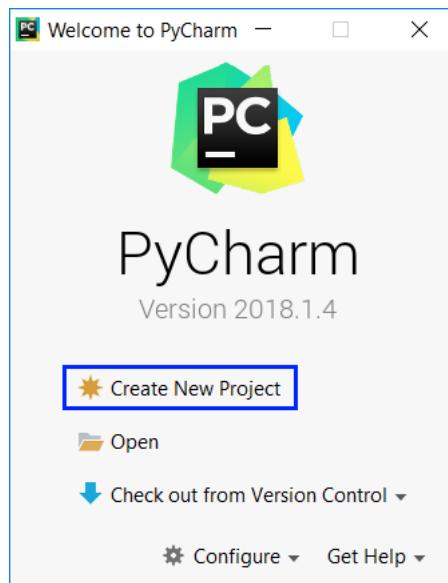


За разлика от конзолните приложения, които четат и пишат данните си във вид на текст на конзолата, уеб приложението имат **уеб базиран потребителски интерфейс**. Уеб приложението се **зареждат от някакъв Интернет адрес** (URL) чрез стандартен уеб браузър. Потребителите пишат входните данни в страница, визуализирана от уеб браузъра, данните се обработват на уеб сървър и резултатите се показват отново в страницата в уеб браузъра. За нашето уеб приложение ще използваме **Flask**, лека Python библиотека, която позволява създаване на **уеб приложения с езика за програмиране Python**.

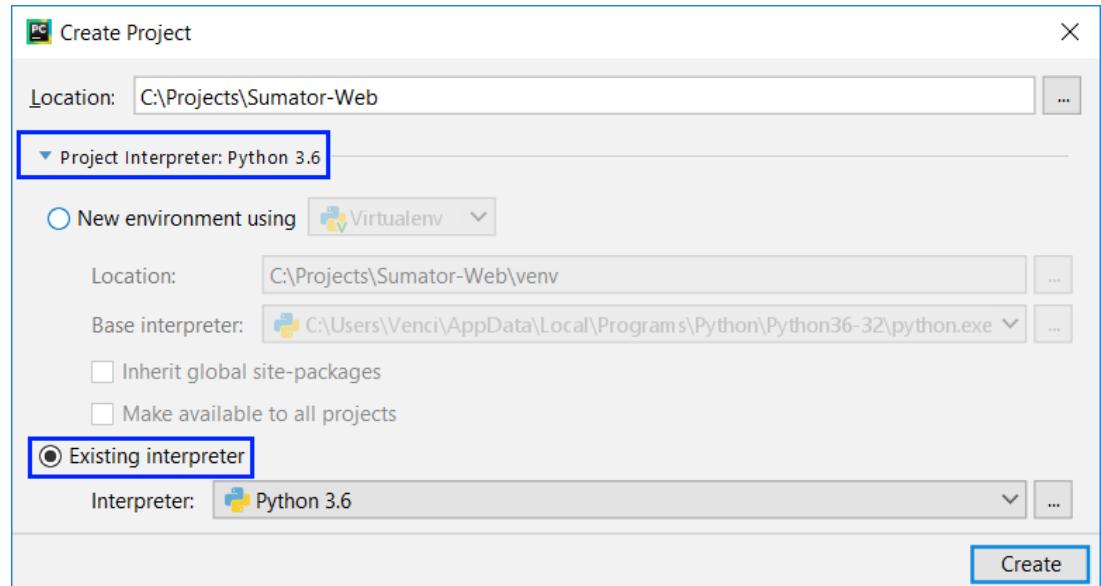
Следват стъпките за имплементация на уеб приложението "Суматор за числа".

## Празно Python решение

В PyCharm създаваме нов Python проект, за да организираме кода от приложението:

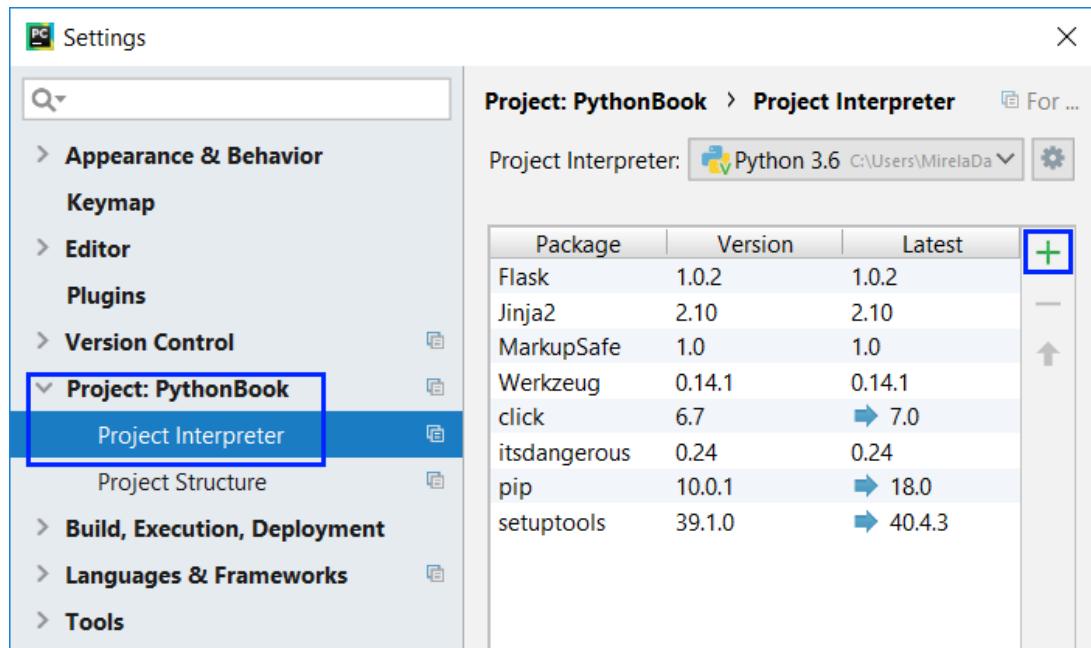


След това, задаваме смислено име на проекта, например "Sumator-Web". Задаваме Python интерпретатора на този по подразбиране:

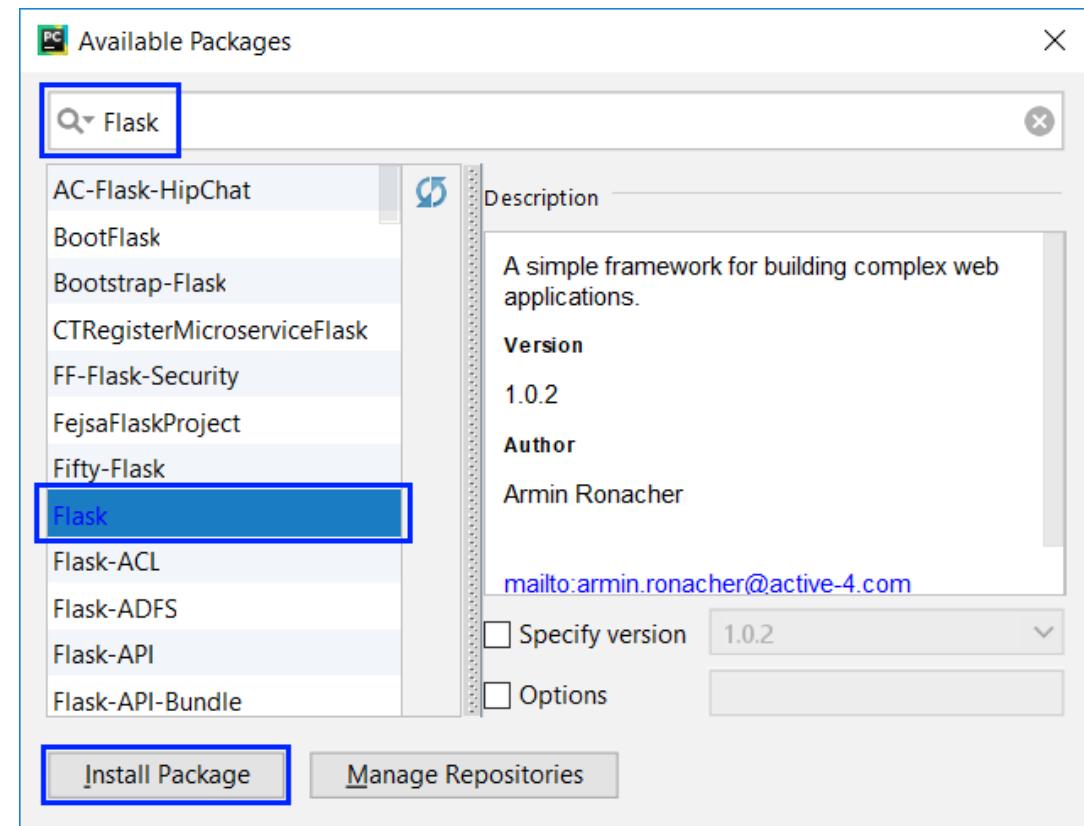


## Инсталиране на Flask

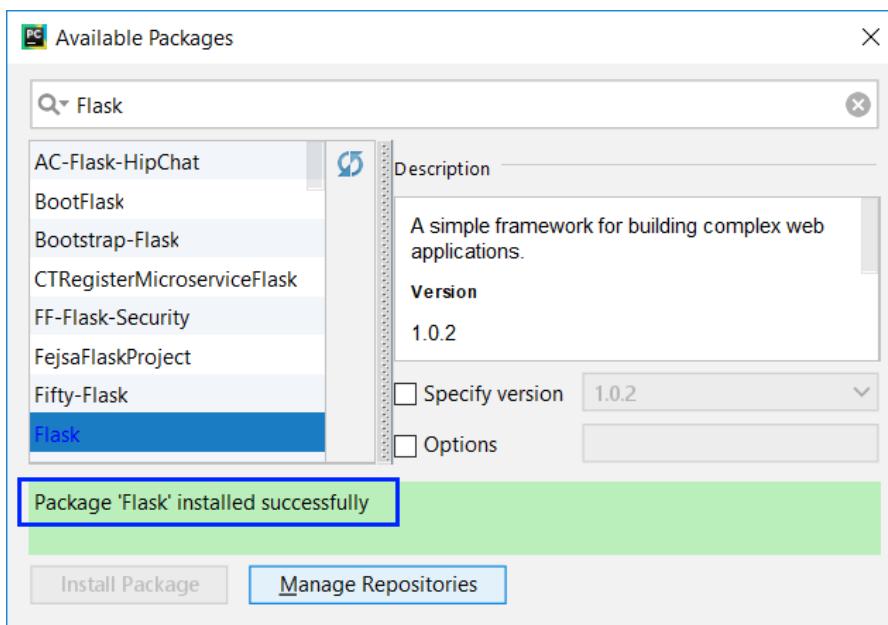
Преди да започнем да пишем код, трябва да инсталираме **Flask**. Отиваме в настройките на PyCharm [File] -> [Settings] и след това в [Project: Sumator-Web] -> [Project Interpreter]. Там кликаме върху бутона **+**:



Търсим **Flask** в прозореца, който излиза и кликаме върху [Install Package]:

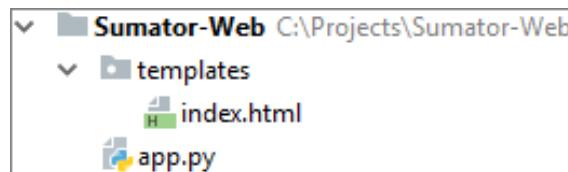


Ако всичко мине успешно, ще получим следното съобщение в същия екран:



## Създаване структурата на проекта

Нека да създадем структурата на нашия проект. За целта създаваме Python файл с име **app.py**, който ще съдържа нашия програмен код. Създаваме папка с име **templates** и в нея HTML файл с име **index.html**, който ще ни помага за изобразяването в браузъра.



## Съставяне логиката на приложението

Сега създаваме контролите на приложението. Целта е да добавим полета за въвеждане на числа, бутон за сумирането им и място, където ще се изобрази резултата. Затова трябва да напишем следния код в **templates/index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sumator</title>
</head>
<body>
    <h2>Sumator</h2>
    <form method="POST" action="/">
        <input type="number" name="firstNumber" />
```

```

<span>+</span>
<input type="number" name="secondNumber" />
<span>=</span>
<span>{{result}}</span>
<input type="submit" value="Calculate" />
</form>
</body>
</html>

```

Този код създава уеб форма **<form>** с поле и бутон [Calculate] за изпращане на данните от формата към сървъра. Обработването на данните ще се случва в нашия Python файл **app.py**. Обърнете внимание на къдравите скоби – те служат за превключване между езика **HTML** и езика **Python** и идват от **Jinja2** синтаксиса за рисуване на динамични уеб страници.

След като напишем следния код в **app.py**, може да тестваме програмата дали работи като я стартираме с **[Ctrl + Shift + F10]** или с десен бутон - **[Run]**:

```

from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')

if __name__ == '__main__':
    app.run()

```

Следва да напишем кода, който ще обработва нашата **заявка**:

```
from flask import Flask, render_template, request
```

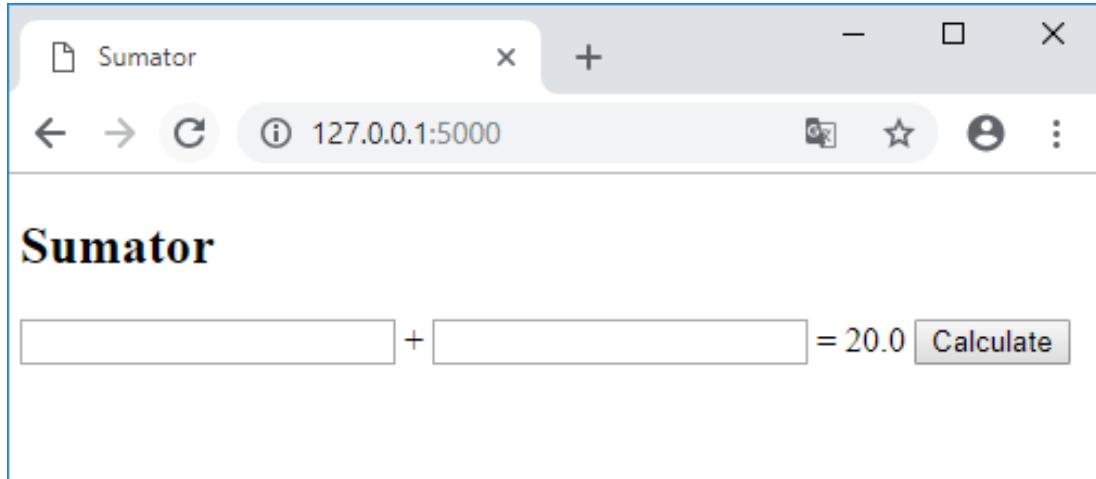
```
app = Flask(__name__)
```

```
@app.route('/', methods=['GET', 'POST'])
def index():
```

```
if request.method == 'GET':
    return render_template('index.html')
elif request.method == 'POST':
    first_value = float(request.form.get('firstNumber'))
    second_value = float(request.form.get('secondNumber'))
    result = eval(str(first_value + second_value))
    return render_template('index.html', result=result)

if __name__ == '__main__':
    app.run()
```

Вече при кликане на бутона за пресмятане, нашата програма ще може да събере двете числа от формата. Нека пуснем програмата отново и да тестваме дали работи коректно. При въвеждане на числата **9** и **11** ще получим верен резултат:



## Обработка на невалидни числа

Ако пробваме с невалидни данни, ще получим грешка:

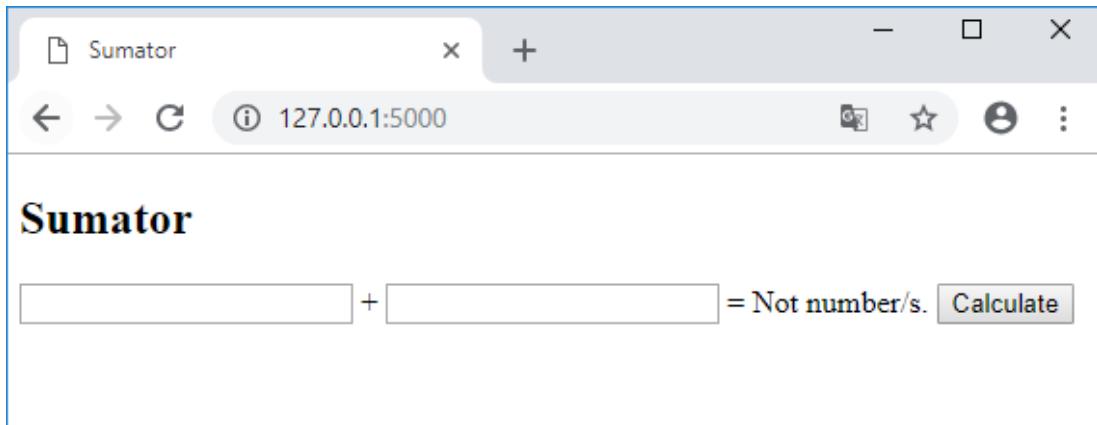


За да се справим с грешката, може да напишем следния код:

```
@app.errorhandler(500)
def page_not_found(e):
    return render_template('index.html', result="Not number/s.")
```

Това не е най-доброто решение за управление на грешки, но ще работи.

Ако пуснем отново програмата и тестваме с **невалидни данни**, ще получим съобщение - **Not number/s.:**



Страшно ли изглежда? **Не се плашете!** Имаме да учим още много, за да достигнем ниво на знания и умения, за да пишем свободно уеб-базирани приложения, като в примера по-горе и много по-големи и по-сложни. Ако не успеете да се справите, продължете спокойно напред. След време ще си спомняте с усмивка колко непонятен и вълнуващ е бил първият ви сблъсък с уеб програмирането. Ако имате проблеми с примера по-горе, питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Целта на горните два примера (графично desktop приложение и уеб приложение) не е да се научите, а да се докоснете по-надълбоко до програмирането, **да разпалите интереса си** към разработката на софтуер и да се вдъхновите да учите здраво. **Имате да учите още много**, но пък е интересно, нали?

# Глава 2.1. Прости пресмятания с числа

В настоящата глава ще се запознаем с някои концепции и програмни техники:

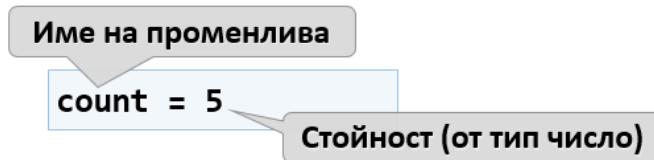
- Как да работим с **типове данни и променливи**, които са ни необходими при обработка на числа и стрингове.
- Как да **изпечатаме** резултат на екрана.
- Как да **четем** потребителски вход.
- Как да извършваме прости **аритметични операции**: събиране, изваждане, умножение, деление, съединяване на стринг.
- Как да **закръгляме** числа.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=qcBWD8ImVRI>.

## Пресмятания в програмирането

За компютрите знаем, че са машини, които обработват данни. Всички **данни** се записват в компютърната памет (RAM памет) в **променливи**. Променливите са именувани области от паметта, които пазят данни от определен тип, например число или текст. Всяка една **променлива** в Python има **име** и **стойност**. Ето как бихме дефинирали една променлива, като едновременно с декларацията ѝ, ѝ присвояваме и стойност:



След тяхната обработка, данните се записват отново в променливи (т.е. някъде в паметта, заделена от нашата програма).

## Типове данни и променливи

В програмирането всяка една променлива съхранява определена **стойност** от даден **тип**. Типовете данни могат да бъдат например: **число**, **буква**, **текст** (стринг), **дата**, **цвят**, **картинка**, **списък** и др. Ето няколко примера за типове данни и стойности за тях:

- **int** – цяло число: 1, 2, 3, 4, 5, ...
- **float** – дробно число: 0.5, 3.14, -1.5, ...
- **str** – текст (низ) от символи: 'a', 'Здрави', 'Hi', 'Beer', ...
- **datetime** – дата: 21-12-2017, 25/07/1995, ...

В езика **Python** типът се определя от стойността, която се присвоява и не се задава изрично при декларирането на променливи (за разлика от C#, Java и C++).

## Печатане на резултат на екрана

За да изпечатаме текст, число или друг резултат на екрана, е необходимо да извикаме вградената функция **print(...)**. С нея можем да принтираме както стойността на променлива, така и директно текст или число:

```
print(9) # печатане на число
print('Hello!') # печатане на текст
msg = 'Hello, Python!'
print(msg) # печатане на стойност на променлива
```

## Четене на потребителски вход – цяло число

За да прочетем цяло число от потребителя, е необходимо да **декларираме променлива** и да използваме вградените функции **input(...)** за четене на текстов ред от конзолата и **int(...)** за преобразуване на текстова стойност към числена:

```
num = int(input())
```

Ако това преобразуване не се направи, за програмата **всяко едно число** ще бъде просто **текст**, с който **не бихме могли да извършваме** аритметични операции. При извикването на **input(...)** можем да подадем подканващо съобщение за потребителя, с което да го насочим какво искаме от него да въведе, примерно:

```
size = int(input('Enter the size = '))
```

## Пример: пресмятане на лице на квадрат със страна a

За пример да вземем следната програма, която прочита **цяло число**, умножава го по него самото (т.е. **вдига го на квадрат**) и отпечатва резултата от умножението. Така можем да пресметнем лицето на квадрат по дадена дължина на страната:

```
a = int(input('a = '))
area = a * a
print('Square area = ', area)
```

Ето как би работила програмата при квадрат с размер на страната 3:

```
Run: square_area
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
a = 3
Square area = 9
Process finished with exit code 0
```

Опитайте да въведете грешно число, например "hello". Ще получите съобщение за **грешка** по време на изпълнение (exception). Това е нормално. По-късно ще разберем как можем да прихващаме такива грешки и да караме потребителят да въвежда число наново.

## Как работи примерът?

Първият ред `a = int(input('a = '))` печата информативно съобщение, което подканва потребителя да въведе страната на квадрата **a**, след това прочита текст (стринг) и го преобразува до цяло число (извършва се т.нар. парсване) чрез функцията `int(...)`. Резултатът се записва в променлива с име **a**.

Следващата команда `area = a * a` записва в нова променлива **area** резултата от умножението на **a** по **a**.

Последният ред `print('Square area = ', area)` отпечатва посочения текст, като до него долепя изчисленото лице на квадрата, който сме записали в променливата **area**.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#0>.

## Четене на поредица числа

Когато искаме да прочетем **няколко** числа от конзолата, ако те са дадени **всяко на отделен ред**, ги четем последователно ето така:

```
num1 = input()
num2 = input()
num3 = input()
print(num1, num2, num3)
```

Ако въведем следния вход:

```
10
20
30
```

ще получим съответно следния резултат:

```
10 20 30
```

Когато искаме да прочетем **няколко** числа от конзолата и те са дадени **заедно на един ред**, разделени с интервал, можем да използваме следна конструкция:

```
num1, num2, num3 = map(int, input().split())
print(num1 + num2 + num3)
```

Ако въведем следния вход:

```
100 200 300
```

ще получим съответно следния резултат:

```
600
```

Как работи горният код? Чрез `.split(...)` разделяме елементите на въведенния текстов ред по разделител интервал. Ако въведем горния вход, ще получим 3 елемента: '100', '200' и '300'. След това чрез функцията `map(int, elements)` преобразуваме поредицата елементи от текст към число.

## Четене на дробно число

За да прочетем потребителски вход под формата на **дробно число**, отново е необходимо да **декларираме променлива**, но този път ще използваме функцията `float(...)`, която преобразува прочетената текстова стойност към числена:

```
num = float(input())
```

## Пример: прехвърляне от инчове в сантиметри

Да напишем програма, която чете дробно число в инчове и го обръща в сантиметри:

```
inches = float(input('Inches = '))
centimeters = inches * 2.54
print('Centimeters = ', centimeters)
```

Да стартираме програмата и да се уверим, че при подаване на стойност в инчове, получаваме коректен резултат в сантиметри:

```
Run:  inches_to_centimeters x
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Inches = 5
Centimeters = 12.7
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#1>.

## Четене на вход под формата на текст

За да прочетем текст (стринг) от конзолата, можем да **декларираме нова променлива** и да използваме стандартната команда за четене от конзолата:

```
str = input()
```

## Пример: поздрав по име

Да напишем програма, която въвежда името на потребителя и го поздравява с текста "Hello, {име}!".

```
name = input()
print('Hello, ', end=' ')
print(name, end='!')
```

По подразбиране вградената функция **print(...)** печата резултат и отива на следващия ред. Това е така, защото **print(...)** използва параметъра **end**, който по подразбиране има стойност **\n** (нов ред). За да останем на същия ред, може да променим стойността на **end=' '**.

Ето и резултата, ако извикаме функцията с името "Иван":

```
Run: ⚡ greeting_by_name
▶ ↑ C:\Projects\SimpleCalculations\venv\Scripts\python.exe
└ ↓
  █ Иван
  █ Hello, Иван!
  █ Process finished with exit code 0
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#2>.

## Съединяване на текст и числа

При печат на текст, числа и други данни, **можем да ги съединим**, като използваме шаблони **{0}**, **{1}**, **{2}** и т.н. В програмирането тези **шаблони** се наричат **placeholders**.

В Python използваме вградения в текстовия тип данни метод **.format(...)**, чрез който изреждаме променливите, които искаме да се поставят на мястото на шаблоните (плейсхолдърите). Пример: въвеждаме **име** и **фамилия** на курсист, неговата **възраст** и от **град** е и отпечатваме текст в стил "You are {име} {фамилия}, a {възраст}-years old person from {град}.". Ето едно решение с текстови шаблони:

```
first_name = input()
last_name = input()
age = int(input())
town = input()

print('You are {0} {1}, a {2}-years old person from {3}.'
      .format(first_name, last_name, age, town))
```

Ето резултата, който ще получим, след изпълнение на този пример:

```
Run: concatenate_data
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Ivan
Ivanov
25
Sofia
You are Ivan Ivanov, a 25-years old person from Sofia.

Process finished with exit code 0
```

Обърнете внимание как всяка една променлива трябва да бъде подадена в реда, в който искаме да се печата. По същество, шаблонът (placeholder) приема променливи от всякакъв вид и това създава голямо удобство при печатане.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#3>.

## Още за отпечатването на форматиран текст

Възможно е един и същ номер на шаблон да се използва по няколко пъти и не е задължително шаблоните за функцията `.format(...)` да са номерирани поредно. Ето пример:

```
print('{1} + {1} = {0}'.format(1+1, 1))
```

Резултатът е:

```
1 + 1 = 2
```

## Аритметични операции

Да разгледаме базовите аритметични операции в програмирането.

### Събиране на числа: оператор `+`

Можем да събираме числа с оператора `+`:

```
a = 5
b = 7
sum = a + b # резултатът е 12
```

### Изваждане на числа: оператор `-`

Изваждането на числа се извършва с оператора `-`:

```
a = int(input())
b = int(input())

result = a - b
print(result)
```

Ето резултата от изпълнението на програмата (при числа 10 и 3):

## Умножение на числа: оператор `*`

За умножение на числа използваме оператора `*`:

```
a = 5
b = 7
product = a * b # 35
```

## Деление на числа: оператор `/`

Делението на числа се извършва с оператора `/`.

**Забележка:** Делението на 0 предизвиква грешка по време на изпълнение (runtime exception) - `ZeroDivisionError`.

Ето няколко примера за използване на оператора за делене:

```
print(10 / 2.5) # Резултат: 4
print(10 / 4)   # Резултат: 2.5
print(10 / 6)   # Резултат: 1.666666666666667

print(5 / 0)    # Резултат: ZeroDivisionError: division by zero
print(-5 / 0)   # Резултат: ZeroDivisionError: division by zero
print(0 / 0)    # Резултат: ZeroDivisionError: division by zero
print(2.5 / 0)  # Резултат: ZeroDivisionError: float division by
                 # zero
```

## Степенуване на числа: оператор `**`

За повдигане на степен в Python използваме оператора `**`:

```
print(2 ** 10)      # 1024
print(2 ** 20)      # 1048576
print(2 ** 30)      # 1073741824
print(2 ** 64)      # 18446744073709551616
print(2 ** 128)     # 340282366920938463463374607431768211456
print(2.5 ** 1.5)   # 3.952847075210474
```

Както се вижда от примера, Python работи с лекота с големи целочислени стойности, без специални усилия. Ето един по-голям пример:

```
print(2 ** 1024)
```

Резултатът е следното **голямо цяло число**:

```
1797693134862315907729305190789024733617976978942306572734300811577
3267580550096313270847732240753602112011387987139335765878976881441
6622492847430639474124377767893424865485276302219601246094119453082
9520850057688381506823424628814739131105408272371633505106845862982
39947245938479716304835356329624224137216
```

## Съединяване на текст и число

Операторът **+** освен за събиране на числа служи и за съединяване на текст (долепяне на два символни низа един след друг). В програмирането съединяване на текст с още текст или с число наричаме "**конкатенация**". Ето как можем да съединяваме текст и число с оператора **+**:

```
first_name = 'Maria'
last_name = 'Ivanova'
age = 19

str = first_name + ' ' + last_name + ' @ ' + str(age)
print(str) # Maria Ivanova @ 19
```

В Python не можем директно да долепяме (конкатенираме) число към даден текст и затова първо превръщане числото към текст с функцията **str(...)**.

Ето още един пример:

```
a = 1.5
b = 2.5

sum = 'The sum is: ' + str(a) + str(b)
print(sum) # The sum is 1.52.5
```

Забелязвате ли нещо странно? Може би очаквахте числата **a** и **b** да се сумират? Всъщност конкатенацията работи последователно отляво надясно и горният

резултат е абсолютно коректен. Ако искаме да сумираме числата, ще трябва да ползваме **скоби**, за да променим реда на изпълнение на операциите:

```
a = 1.5
b = 2.5

sum = 'The sum is: ' + str(int(a + b))
print(sum) # The sum is: 4
```

## Повтаряне на текст: оператор \*

Операторът **\*** може да се използва за **повтаряне на даден текст няколко пъти**:

```
text = 'hi'
print(3 * text) # hihihi
```

Тази особеност на оператора за умножение може да доведе до следния грешен **резултат**, когато не съобразим типа на умножаваната стойност:

```
count = input() # Enter 20
print(5 * count) # 2020202020, not 100
```

Ако искаме да умножим по 5 число, прочетено от входа, трябва първо да преобразуване към число с функцията **int()**:

```
count = int(input()) # Enter 20
print(5 * count) # 100
```

## Отпечатване на форматиран текст в Python

В езика Python има **няколко начина** да отпечатаме **форматиран текст**, т.е. текст, смесен с числа, стойности на променливи и изрази. Вече се сблъскахме с някои от тях, но нека разгледаме въпроса по-детайлно.

### Печатане с изброяване със запетайки

При печатане с **print(...)** можем да изброяваме няколко стойности със запетайки:

```
width = 5
height = 7
print('width =', width, '; height =', height, '; area =', width
* height)
```

Резултатът е следният:

```
width = 5 ; height = 7 ; area = 35
```

Както се вижда от примера, при такова отпечатване всеки две стойности от изброените се отделят с **интервал**.

## Съединение на текст с оператора +

Вече знаем как да съединяваме текст и числа с оператора **+**. Използвайки **+**, горният пример може да се напише по следния начин:

```
width = 5
height = 7
print('width = ' + str(width) + ' ; height = ' + str(height) +
      ; area = ' + str(width * height))
```

Резултатът е същият като в предния пример:

```
width = 5 ; height = 7 ; area = 35
```

Понеже в Python текст не може да се съединява директно с числа, те трябва да се преобразуват първо към текст чрез функцията **str(num)**.

## Форматиращи низове %d, %s, %f

Печатането с шаблони може да се извърши и чрез **форматиращи низове** (които се ползват в езици като C и Java). Ето пример как става това:

```
width = 5
height = 7
text = "area"
print('width = %d ; height = %d ; %s = %d' % (width, height,
                                                text, width * height))
```

В горния пример използваме оператора **%**, който замества в **текстов шаблон** стойности, подадени като поредица от елементи в скоби. Използват се следните основни **placeholders** (форматиращи низове): **%d** обозначава цяло число, **%f** обозначава дробно число, **%s** обозначава текст. Резултатът от горния код е същият като в предходните примери:

```
width = 5 ; height = 7 ; area = 35
```

Ако е подаден грешен форматиращ низ, може да получим грешка.

При форматирането на **дробни числа** можем да закръглеме до определен брой цифри след десетичната точка, например с форматиращ низ **%.2f** отпечатваме дробно число с 2 знака след десетичната точка:

```
print('price = %.2f \nVAT = %.3f' % (1.60, 1.60 * 0.2))
```

Резултатът от горния код е следният:

```
price = 1.60
VAT = 0.320
```

В горния пример използваме специалния символ **\n**, който означава „отиди на нов ред“. Аналогично специалният символ **\b** връща курсора с една позиция назад (изтрива последния отпечатан символ) и съответно следния код:

```
print('ab\b\bc')
```

отпечатва следния резултат:

```
ac
```

## Форматиране с .format(...)

Вече разгледахме как можем да форматираме текст и числа чрез **.format(...)** с използването на **номерирани шаблони {0}, {1}, {2}** и т.н. Въсъщност, когато числова номерация не е необходима, можем да напишем само **{}**, за да създадем **placeholder**. Ето пример как можем да ползваме форматиране с **текст.format(...)**, за да отпечатаме резултата от предходния пример:

```
width = 5
height = 7
print('width = {} ; height = {} ; area = {}'.format(width,
height, width * height))
```

Резултатът е отново същият:

```
width = 5 ; height = 7 ; area = 35
```

## Форматиране с f-string

Може би най-лесното, най-интуитивното и най-краткото за писане **форматиране на текст и числа** в Python е чрез **f-string синтаксиса**:

```
width = 5
height = 7
print(f'width = {width} ; height = {height} ; area = {width *
height}')
```

Използването на **f-string форматиране** е много лесно: поставяме префикс **f** пред стринга и в него вмъкваме на произволни позиции стойности на променливи и изрази във фигурни скоби **{expression}**. В горния пример имаме три израза: **{width}**, **{height}** и **{width \* height}**, които се изчисляват и се заместват с текстовата им стойност. Можем да ползваме текстови изрази, целочислени изрази, изрази с дробни числа и всякакви други изрази.

Форматирането с **f-string** в Python е **препоръчителният начин** за печатане на форматиран текст, защото ползва най-кратък синтаксис и кодът е най-лесно разбираем се чете най-лесно.

## Числени изрази

В програмирането можем да пресмятаме и **числови изрази**, например:

```
expr = (3 + 5) * (4 - 2) # 16
```

В сила е стандартното правило за приоритетите на аритметичните операции: **умножение и деление се извършват винаги преди събиране и изваждане**. При наличие на **израз** в скоби, той се изчислява пръв, но ние знаем всичко това от училищната математика.

### Пример: изчисляване на лице на трапец

Да напишем програма, която въвежда дължините на двете основи на трапец и неговата височина (по едно дробно число на ред) и пресмята **лицето на трапеца** по стандартната математическа формула:

```
b1 = float(input())
b2 = float(input())
h = float(input())

area = (b1 + b2) * h / 2
print('Trapezoid area = ' + str(area))
```

Тъй като искаме програмата да работи, както с цели, така и с дробни числа, използваме **float(...)**. Ако стартираме програмата и въведем за страните съответно **3, 4 и 5**, ще получим следния резултат:

```
Run: trapeziod_area
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
3
4
5
Trapezoid area = 17.5
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#4>.

## Закръгляне на числа

Понякога, когато работим с дробни числа, се налага да приведем числата към еднотипен формат. Това привеждане се нарича **закръгляне**. Езикът **Python** ни предоставя няколко метода за закръгляне на числа:

- **`math.ceil(...)`** – закръгляне нагоре, до следващо (по-голямо) цяло число:

```
up = math.ceil(23.45) # up = 24
```

- **`math.floor(...)`** – закръгляне надолу, до предишно (по-малко) цяло число:

```
down = math.floor(45.67) # down = 45
```

- **`round(...)`** – закръглянето се извършва по **основното правило за закръгляване** - ако десетичната част е по-малка от 5, закръглението е надолу и обратно, ако е по-голяма от 5 - нагоре:

```
round(5.439) # 5
round(5.539) # 6
```

- **`round(..., [брой символи след десетичната запетая])`** – закръгляне до най-близко число с определен на брой символи след десетичната запетая:

```
round(123.456, 2) # 123.46
round(123, 2) # 123.0
round(123.456, 0) # 123.0
round(123.512, 0) # 124.0
```

## Пример: периметър и лице на кръг

Нека напишем програма, която при въвеждане **радиуса r** на кръг **изчислява лицето и периметъра** на кръга / окръжността. Да си припомним формулите:

- Лице =  $\pi * r * r$
- Периметър =  $2 * \pi * r$
- $\pi \approx 3.14159265358979323846\dots$

```
import math

r = float(input('Enter circle radius => r = '))

area = math.pi * r * r
perimeter = 2 * math.pi * r
```

```
print('Area = ', area)
print('Perimeter = ', perimeter)
```

Нека изprobваме програмата с радиус **r = 10**:

```
Run: circle_area_and_perimeter ×
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Enter circle radius => r = 10
Area = 314.1592653589793
Perimeter = 62.83185307179586

Process finished with exit code 0
```

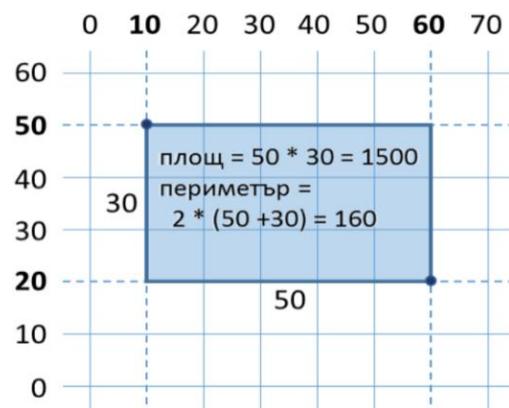
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#5>.

## Пример: лице на правоъгълник в равнината

Правоъгълник е зададен с координатите на два от своите два срещуположни ъгъла. Да напише програма за пресмятане на площта и периметъра му.

В тази задача трябва да съобразим, че ако от по-големия **x** извадим по-малкия **x**, ще получим дължината на правоъгълника. Аналогично, ако от по-големия **y** извадим по-малкия **y**, ще получим височината на правоъгълника. Остава само да умножим двете страни.



Ето примерна имплементация на описаната логика:

```
x1 = float(input())
y1 = float(input())
x2 = float(input())
y2 = float(input())

width = max(x1, x2) - min(x1, x2)
height = max(y1, y2) - min(y1, y2)

area = width * height
perimeter = 2 * (width + height)

print('Area = ', area)
print('Perimeter = ', perimeter)
```

Използваме функцията `max(a, b)`, за намиране на по-голямата измежду стойностите **a** и **b** и аналогично `min(a, b)` - за намиране на по-малката от двете стойности.

При стартиране на програмата със стойностите от координатната система в условието, получаваме следния резултат:

```
Run: rectangle_area
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
60
20
10
50
Area = 1500.0
Perimeter = 160.0

Process finished with exit code 0
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#6>.

## Какво научихме от тази глава?

Да резюмираме какво научихме от тази глава на книгата:

- Въвеждане на текст: `str = input()`.
- Въвеждане на цяло число: `num = int(input())`.
- Въвеждане на дробно число: `num = float(input())`.
- Извършване на пресмятания с числа и използване на съответните аритметични оператори [+,-,\*,/,:]: `sum = 5 + 3`.
- Извеждане на текст по шаблон на конзолата: `print('{0} + {1} = {2}'.format(3, 5, 3 + 5))`.

## Упражнения: прости пресмятания

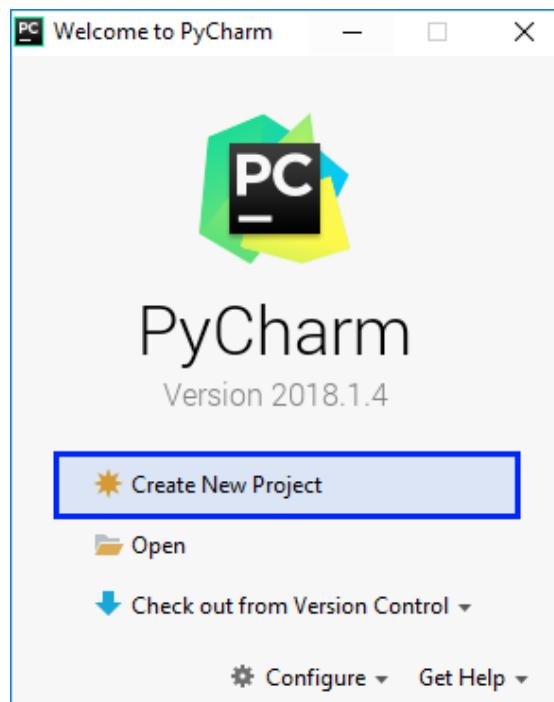
Нека затвърдим наученото в тази глава с няколко задачи.

### Създаване на нов проект в PyCharm

Създаваме нов проект в PyCharm (от [Create New Project] или [File] -> [New Project]), за да организираме по-добре задачите за упражнение. Целта на този проект е да съдържа по един Python файл за всяка задача от упражненията:

- Стартираме PyCharm.

- Създаваме нов проект: [Create New Project] (или [File] -> [New Project]).

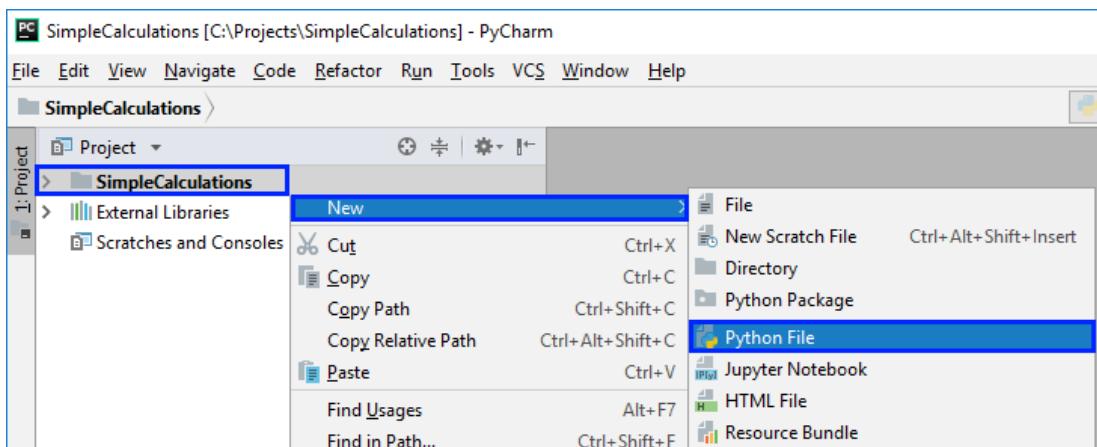


## Задача: пресмятане на лице на квадрат

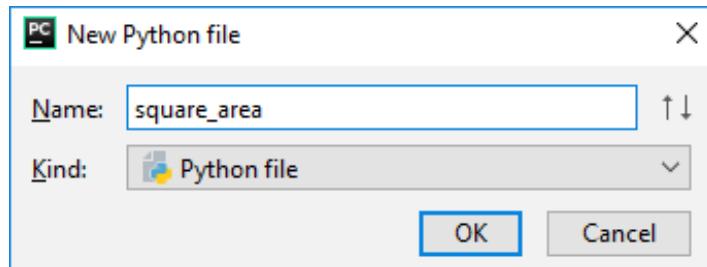
Първата задача от тази тема е следната: да се напише конзолна програма, която въвежда цяло число **a** и пресмята лицето на квадрат със страна **a**. Задачата е тривиално лесна: прочитаме въведеното число, умножаваме го само по себе си и печатаме получения резултат на конзолата.

## Насоки и подсказки

Създаваме нов файл в съществуващия PyCharm проект. Кликваме с десен бутон на мишката върху SimpleCalculations и избираме [New] -> [Python File]:



Ще се отвори диалогов прозорец за избор на името на файла. Наименуваме го "square\_area":



Вече имаме проект с един файл в него. Остава да напишем кода за решаване на задачата. За целта отиваме във файла и пишем следния код:

```
a = int(input('a = '))
area = a * a
print('Square area = ', area)
```

Кодът прочита цяло число чрез `a = int(input('a = '))`, след това изчислява `area = a * a` и накрая печата стойността на променливата `area`. Стартураме програмата с [Ctrl + Shift + F10] или с десен бутон - [Run], и я тестваме с различни входни стойности:

```
Run: python square_area.py
▶ C:\Projects\SimpleCalculations\venv\Scripts\python.exe
a = 5
Square area = 25
Process finished with exit code 0
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#0>. Трябва да получите 100 точки (напълно коректно решение):

### 01. Square Area

```
1 a = int(input('a = '))
2 area = a * a
3 print('Square area = ', area)
4
```

Allowed working time: 0.100 sec  
 Allowed memory: 16.00 MB  
 Size limit: 16.00 KB  
 Checker: Numbers Checker

Python code ▾ Submit

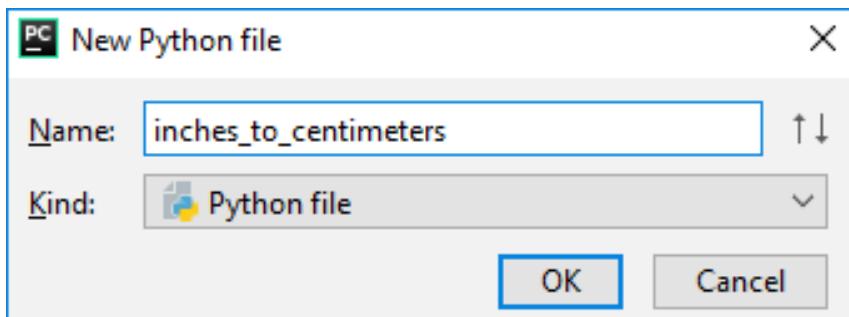
Submissions		
1		
Points	Time and memory used	Submission date
100 / 100	Memory: 8.24 MB Time: 0.056 s	18:26:13 21.08.2018
		<a href="#">Details</a>

### Задача: от инчове към сантиметри

Да се напише програма, която чете от конзолата число (не непременно цяло) и преобразува числото от **инчове в сантиметри**. За целта умножава инчовете по 2.54 (защото 1 инч = 2.54 сантиметра).

#### Насоки и подсказки

Първо създаваме **нов Python файл** в проекта "SimpleCalculations". За целта кликаме с десен бутон на мишката върху решението SimpleCalculations и избираме [New] -> [Python File]. Задаваме име "**inches\_to\_centimeters**":



Следва да напишем кода на програмата:

```
inches = float(input('Inches = '))
centimeters = inches * 2.54
print('Centimeters = ', centimeters)
```

Стартираме програмата с [Ctrl + Shift + F10] или с десен бутон - [Run]:

```

1  inches = float(input('Inches = '))
2  centimeters = inches * 2.54
3  print('Centimeters = ', centimeters)
4
5
    
```

Copy Reference Ctrl+Alt+Shift+C  
 Paste Ctrl+V  
 Paste from History... Ctrl+Shift+V  
 Paste Simple Ctrl+Alt+Shift+V  
 Column Selection Mode Alt+Shift+Insert  
 Find Usages Alt+F7  
 Refactor >  
 Folding >  
 Go To >  
 Generate... Alt+Insert  
**Run 'inches\_to\_centimeters' Ctrl+Shift+F10**  
 Debug 'inches\_to\_centimeters'  
 Save 'inches\_to\_centimeters'  
 Show in Explorer  
 Open in terminal  
 Local History >  
 Run File in Console  
 Compare with Clipboard  
 File Encoding  
 Create Gist...

Да тестваме с дробни числа, например с 2.5:

```

Run: inches_to_centimeters x
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Inches = 2.5
Centimeters = 6.35
Process finished with exit code 0

```

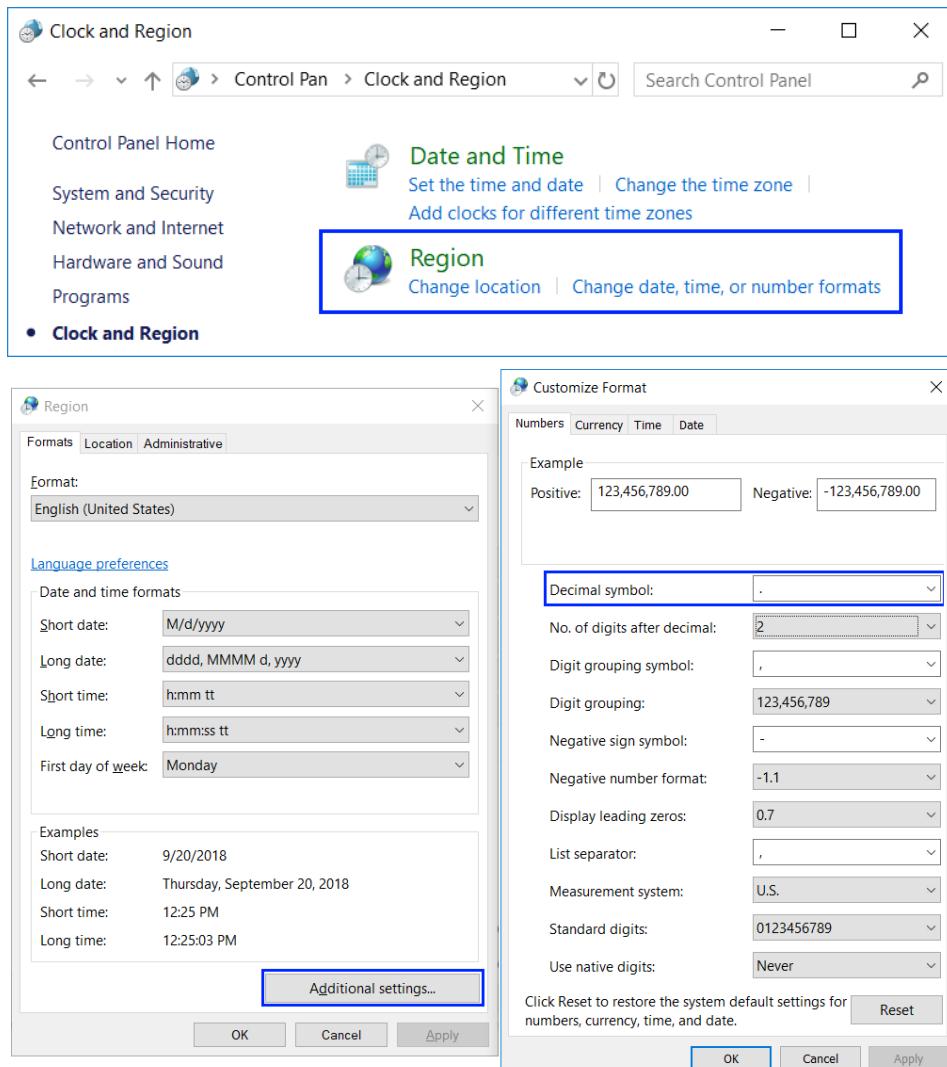


В зависимост от регионалните настройки на операционната система, е възможно вместо **десетична точка** (US настройки) да се използва **десетична запетая** (BG настройки).

Ако програмата очаква десетична точка и бъде въведено число с десетична запетая или обратното (бъде въведена десетична точка, когато се очаква десетична запетая), ще се получи следната грешка:

```
Run: inches_to_centimeters x
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Inches = 2,5
Traceback (most recent call last):
  File "C:/Projects/SimpleCalculations/inches_to_centimeters.py",
    line 1, in <module>
      inches = float(input('Inches = '))
ValueError: could not convert string to float: '2,5'
Process finished with exit code 1
```

Препоръчително е да променим настройките на компютъра си, така че да се използва десетична точка. В Windows това става от контролния панел:



## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#1>.

Решението би трябвало да бъде прието като напълно коректно:

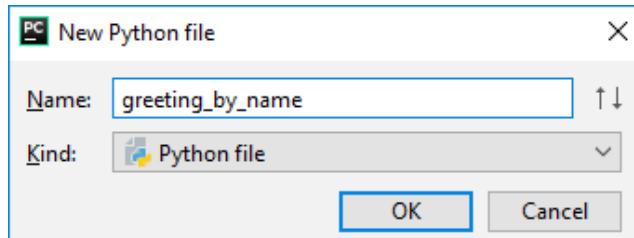
Submissions		
	1	
Points	Time and memory used	Submission date
100 / 100	Memory: 8.27 MB Time: 0.056 s	18:40:49 21.08.2018
	1	

### Задача: поздрав по име

Да се напише програма, която чете от конзолата име на човек и отпечатва **Hello, <name>!**, където **<name>** е въведеното преди това име.

#### Насоки и подсказки

Първо създаваме нов Python файл с име "**greeting\_by\_name**" в проекта "SimpleCalculations":



Следва да напишем кода на програмата. Ако се затруднявате, може да ползвате примерния код по-долу:

```
name = input()
print('Hello, ', end=' ')
print(name, end=' !')
```

Стартираме програмата с [Ctrl + Shift + F10] или с десен бутон - [Run], за да тестваме дали работи коректно:

```
Run:  greeting_by_name
▶ ↑ ↓ ⏪ ⏩ ⏴ ⏵
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Svetlin Nakov
Hello, Svetlin Nakov!
Process finished with exit code 0
```

#### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#2>.

## Задача: съединяване на текст и числа

Напишете програма, която прочита от конзолата име, фамилия, възраст и град и печата съобщение от следния вид: **You are <firstName> <lastName>, a <age>-years old person from <town>**.

### Насоки и подсказки

Добавяме към текущия PyCharm проект още един Python файл с име "**concatenate\_data**". Пишем кода, който чете входните данни от конзолата:

```
first_name = input()
last_name = input()
age = int(input())
town = input()
```

Кодът, който отпечатва описаното в условието на задачата съобщение, е целенасочено размазан и трябва да се допише от читателя:

Следва да се тества решението локално с [Ctrl + Shift + F10] или с десен бутон - [Run], и въвеждане на примерни входни данни.

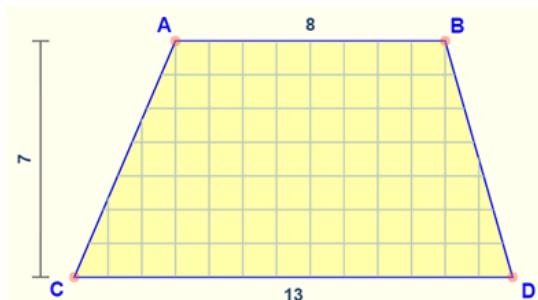
### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#3>.

## Задача: лице на трапец

Напишете програма, която чете от конзолата три числа  $b_1$ ,  $b_2$  и  $h$  и пресмята лицето на трапец с основи  $b_1$  и  $b_2$  и височина  $h$ . Формулата за лице на трапец е  $(b_1 + b_2) * h / 2$ .

### Примерен вход и изход



Вход	Изход
8	
13	73.5
7	

Вход	Изход
12	
8	50
5	

На фигурата по-горе е показан трапец със страни 8 и 13 и височина 7. Той има лице  $(8 + 13) * 7 / 2 = 73.5$ .

## Насоки и подсказки

Отново трябва да добавим към текущия PyCharm проект още един Python файл с име **"trapezoid\_area"** и да напишем кода, който чете **входните данни** от конзолата, пресмята лицето на трапеца и го отпечатва. Кодът на картинката е нарочно размазан, за да помисли читателят върху него и да го допише сам:

```
b1 = float(input())
```

```
print('Trapezoid area = ' + str(area))
```

Тествайте решението локално с [Ctrl + Shift + F10] или с десен бутон - [Run], и въвеждане на примерни данни.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#4>.

## Задача: периметър и лице на кръг

Напишете програма, която чете от конзолата **число r** и пресмята и отпечатва лицето и периметъра на кръг/окръжност с радиус **r**.

### Примерен вход и изход

Вход	Изход	Вход	Изход
3	Area = 28.2743338823081 Perimeter = 18.8495559215388	4.5	Area = 63.6172512351933 Perimeter = 28.2743338823081

## Насоки и подсказки

За изчисленията можете да използвате следните формули:

- **area = math.pi \* r \* r.**
- **perimeter = 2 \* math.pi \* r.**

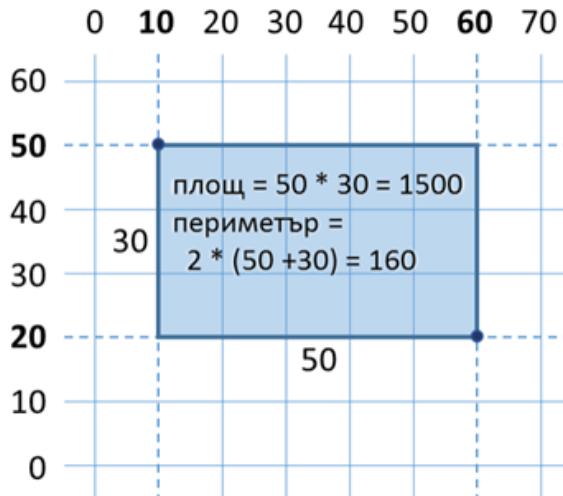
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#5>.

## Задача: лице и периметър на правоъгълник в равнината

Правоъгълник е зададен с **координатите** на два от своите срещуположни ъгъла  $(x_1, y_1) - (x_2, y_2)$ . Да се пресметнат **площта и периметъра** му. **Входът** се чете от конзолата. Числата **x1, y1, x2 и y2** са дадени по едно на ред. **Изходът** се извежда

на конзолата и трябва да съдържа два реда с по една число на всеки от тях – лицето и периметъра.



#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
60		30		600.25	350449.6875
20	1500	40	2000	500.75	2402
10	160	70	180	100.50	
50		-10		-200.5	

#### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#6>.

#### Задача: лице на триъгълник

Напишете програма, която чете от конзолата **страна** и **височина** на триъгълник и пресмята неговото лице. Използвайте **формулата** за лице на триъгълник:  $\text{area} = \text{a} * \text{h} / 2$ . Закръглете резултата до 2 цифри след десетичния знак използвайки функцията **round(area, 2)**.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
20 30	Triangle area = 300	15 35	Triangle area = 262.5
7.75 8.45	Triangle area = 32.74	1.23456 4.56789	Triangle area = 2.82

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#7>.

### Задача: конзолен конвертор - от градуси °C към градуси °F

Напишете програма, която чете градуси по скалата на Целзий ( $^{\circ}\text{C}$ ) и ги преобразува до градуси по скалата на Фаренхайт ( $^{\circ}\text{F}$ ). Потърсете в Интернет подходяща [формула](#), с която да извършите изчисленията. Закръглете резултата до 2 символа след десетичния знак.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
25	77	0	32	-5.5	22.1	32.3	90.14

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#8>.

### Задача: конзолен конвертор - от радиани в градуси

Напишете програма, която чете ъгъл в [радиани](#) (`rad`) и го преобразува в [градуси](#) (`deg`). Потърсете в Интернет подходяща формула. Числото  $\pi$  в Python програмите е достъпно чрез `math.pi` като преди това трябва да реферираме [библиотеката math](#) чрез `import math`. Закръглете резултата до най-близкото цяло число използвайки метода `round()`.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
3.1416	180	0.7854	45
6.2832	360	0.5236	30

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1047#9>.

### Задача: конзолен конвертор - USD към BGN

Напишете програма за конвертиране на щатски долари (USD) в български лева (BGN). Закръглете резултата до 2 цифри след десетичния знак. Използвайте фиксиран курс между долар и лев: 1 USD = 1.79549 BGN.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
20	35.91 BGN	100	179.55 BGN		
				12.5	22.44 BGN

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1047#10>.

## Задача: \* конзолен междувалутен конвертор

Напишете програма за **конвертиране** на парична сума от една валута в друга. Трябва да се поддържат следните валути: BGN, USD, EUR, GBP. Използвайте следните фиксираны валутни курсове:

Курс	USD	EUR	GBP
1 BGN	1.79549	1.95583	2.53405

Входът е сума за конвертиране, входна валута и изходна валута. Изходът е едно число – преобразуваната сума по посочените по-горе курсове, закръглен до **2 цифри** след десетичната точка.

## Примерен вход и изход

Вход	Изход	Вход	Изход
20 USD BGN	35.91 BGN	12.35 EUR GBP	9.53 GBP
100 BGN EUR	51.13 EUR	150.35 USD EUR	138.02 EUR

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1047#11>.

## Задача: \*\* пресмятане с дати - 1000 дни на Земята

Напишете програма, която въвежда **ръждена дата** във формат **dd-MM-уууу** и пресмята датата, на която се навършват **1000 дни** от тази ръждена дата и я отпечатва в същия формат.

## Примерен вход и изход

Вход	Изход
25-02-1995	20-11-1997
07-11-2003	02-08-2006

Вход	Изход
14-06-1980	10-03-1983
01-01-2012	26-09-2014

Вход	Изход
30-12-2002	24-09-2005

## Насоки и подсказки

- Потърсете информация за **datetime** в Python и по-конкретно разгледайте методите **strptime(str, format)**, **timedelta(days=n)**. С тяхна помощ може да решите задачата, без да е необходимо да изчислявате дни, месеци и високосни години.
- Не печатайте нищо допълнително на конзолата освен изискваната дата!

## Тестване в Judge системата

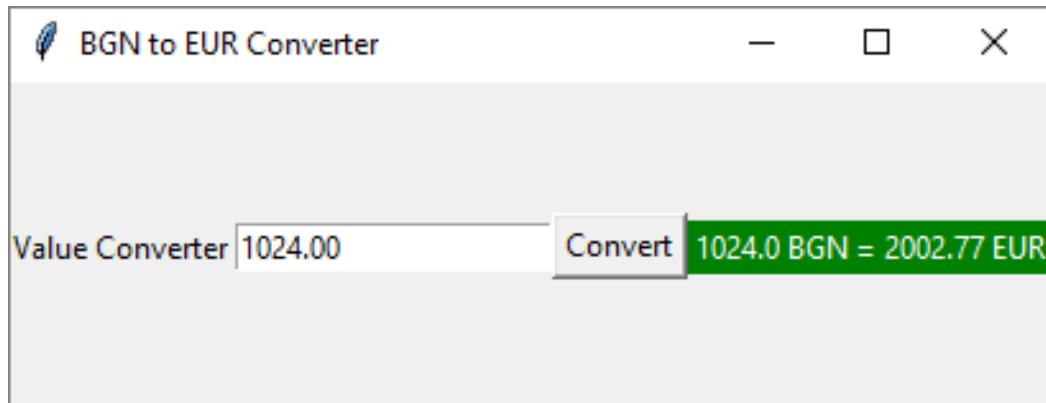
Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1047#12>.

## Графични приложения с числови изрази

За да упражним работата с променливи и пресмятания с оператори и числови изрази, ще направим нещо интересно: ще разработим **настолно приложение** с графичен потребителски интерфейс (GUI). В него ще използваме пресмятания с дробни числа.

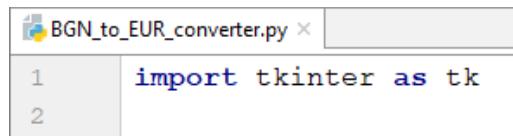
### Графично приложение: конвертор от BGN към EUR

От нас се изиска да създадем **графично приложение** (GUI application), което пресмята стойността в **евро(EUR)** на парична сума, зададена в **лева (BGN)**. Използваме курс лева / евро: 1.95583. Ето пример как може да изглежда крайният продукт:



**Забележка:** тази задача излиза извън изучавания в книгата материал и има за цел не да ви научи как да програмирате GUI приложения, а да ви запали интереса към разработката на софтуер. Да се залавяме за работа.

Добавяме към текущия PyCharm проект още един Python файл. Наименуваме го "**BGN\_to\_EUR\_converter**". За да направим графично приложение с Python, ще използваме стандартната библиотека [tkinter](#).



```
1 import tkinter as tk
2
```

Първо създаваме графично приложение (**Application**), което представлява правоъгълна рамка за компоненти (**Frame**). На този етап всичко ще се случва без обяснения, като **на магия**:

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        self.pack()

# create the application
app = Application()
app.master.title("BGN to EUR Converter")

# start the program
app.mainloop()
```

Вече може да добавим компонентите на приложението ни в т.нар. **функция**, която ще извикаме във функцията **init**, или т.нар. **конструктор**:

```
def __init__(self, master=None):
    super().__init__(master)

    self.pack()
    self.create_widgets()
```

Подреждаме следните UI компоненти:

- **Label** – ще ни служи за статично изобразяване на текст.
- **Entry** – ще въвежда сумата за конвертиране.
- **Button** – ще конвертира въведената сума.
- Още един **Label**, който ще показва резултата след конвертиране.

Нашите компоненти се намират във функцията **create\_widgets()**. Добавяме текст за визуализация на първия **Label**, който ни е под името **label\_numberEntry**

е **Entry**, където ще се въвежда сумата за конвертиране. **convertButton** ще хваща събитие и ще изпълнява **команда** (в нашата задача ще извика функцията **convert()**, която ще напишем малко по-късно). **output** е нашия **Label** за показване на резултат след като сме въвели suma и кликнали върху бутон:

```
class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        self.pack()
        self.create_widgets()

    def create_widgets(self):
        # create widgets
        self.label = tk.Label(text="Value Converter")
        self.numberEntry = tk.Entry()
        self.convertButton = tk.Button(text="Convert",
                                       command=self.convert)
        self.output = tk.Label()
```

След като сме инициализирали нашите компоненти е ред да ги визуализираме. Това става лесно чрез вградения метод в **tkinter** – **pack()**:

```
# place widgets
self.label.pack(side="left")
self.numberEntry.pack(side="left")
self.convertButton.pack(side="left")
self.output.pack(side="left")
```

Остана да напишем **кода** (програмната логика) за конвертиране от лева към евро. Това ще го направим във функцията **convert()**:

```
def convert(self):
    entry = self.numberEntry.get()
    value = float(entry)
    result = round(value * 1.95583, 2)

    self.output.config(
        text=str(value) + ' BGN = ' + str(result) + ' EUR',
        bg="green", fg="white")
```

На последния ред подаваме резултата на нашия **Label output** и задаваме цвят на фона (**bg**) и на текста (**fg**).

Стартираме приложението с [Ctrl + Shift + F10] или с десен бутон + [Run], и тестваме дали работи коректно.

С този код ще имаме проблем. Какво ще стане, ако въведем нещо, различно от число?

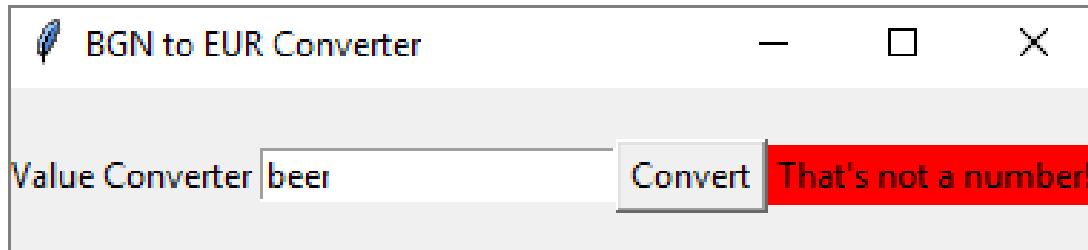
```
Run: BGN_to_EUR_converter x
C:\Projects\SimpleCalculations\venv\Scripts\python.exe
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:/Users/MirelaDamyanova/AppData/Local/Programs/Python/
    Python36-32/lib/tkinter\__init__.py", line 1699, in __call__
    return self.func(*args)
  File "C:/Projects/SimpleCalculations/BGN to EUR converter.py",
    line 26, in convert
      value = float(entry)
ValueError: could not convert string to float: 'beer'
```

Тази грешка можем да я **хванем** и да получаваме user-friendly съобщение в приложението ни. За целта нека променим кода на нашата програмна логика:

```
def convert(self):
    entry = self.numberEntry.get()

    try:
        value = float(entry)
        result = round(value * 1.95583, 2)
        self.output.config(
            text=str(value) + ' BGN = ' + str(result) + ' EUR',
            bg="green", fg="white")
    except ValueError:
        self.output.config(
            text="That's not a number!",
            bg="red", fg="black")
```

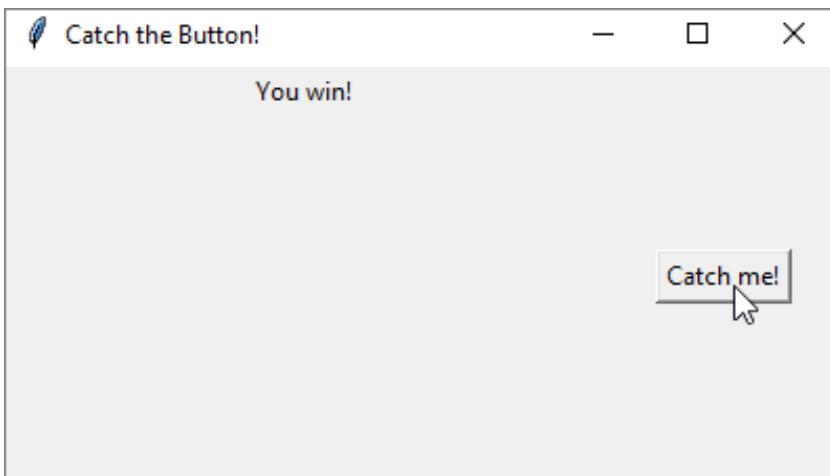
По този начин ние прихващаме грешката в **try** блок и когато въведем нещо, различно от число, ще получаваме съобщение **That's not a number!**.



Накрая **стартираме приложението** с [Ctrl + Shift + F10] или с десен бутон - [Run], и тестваме дали работи коректно.

**Графично приложение: \*\*\* Хвани бутона!**

Създайте забавно графично приложение „хвани бутона“. При преместване на курсора на мишката върху бутона той се премества на случайна позиция. Така се създава усещане, че „бутонът бяга от мишката и е трудно да се хване“. При „хващане“ на бутона се показва съобщение-поздрав.



Ще започнем със същия код от предната задача. Нека променим името на приложението ни на **Catch the Button!** и този път ще зададем размера на прозореца:

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        self.pack()

# create the application
# ... (redacted code)

app.master.minsize(width=400, height=200)

# start the program
# ... (redacted code)
```

Ще имаме следните компоненти:

- **Button** – бутонът, който трябва да хванем.
- **Label** – съобщението-поздрав.

Създаваме бутон с текст "Catch me!" и команда - функцията **on\_click()**, която ще напишем по-късно. Визуализираме компонентите чрез метода **pack()**. Нека и двата компонента са най-отгоре (top):

```
self.pack()
self.create_widgets()

def create_widgets(self):
    # create widgets
    # place widgets
```

В нашата задача ще ползваме т.нар. операция **binding**. Това представлява **хващане** на промяна и изпълняване на определена функция. Чрез този код указваме на програмата ни, че при преместване на курсора на мишката върху бутона, ще се изпълни функцията **on\_enter()**, а при преместването на курсора на мишката извън бутона, ще се изпълни функцията **on\_leave()**:

```
# bind functions to events
# mouse is over -> move button
# leave button -> clear text
self.button.bind("<Enter>", self.on_enter)
self.button.bind("<Leave>", self.on_leave)
```

За преместването на бутона на случайна позиция ще използваме **random**:

```
import random
```

Нека имплементираме трите функции, които съдържат програмната логика в приложението ни:

- **on\_enter(self, event)** – избираме случайни **x** и **y** координати, които ще се променят всеки път, когато преместим курсора на мишката върху бутона; променяме и местоположението на бутона - в нашия пример ще бъде **статично отляво**, но може да направите своя логика и да променяте всеки път посоката на бутона, заедно със случайните числа за координати.
- **on\_leave(self, event)** – когато курсорът на мишката не е върху бутона, нашият поздрав не трябва да се показва; конфигурираме нашия **label** да **няма текст**.
- **on\_click()** – когато кликнем върху бутона, нашия **label** вече е с текст "You win!" .

```
def on_enter(self, event):
    x = random.randrange(0, 100)
    y = random.randrange(0, 100)
    self.button.pack(side="right", padx=x, pady=y)

def on_leave(self, event):
    self.button.pack_forget()

def on_click(self):
    self.button.pack(side="left", padx=y, pady=x)
```

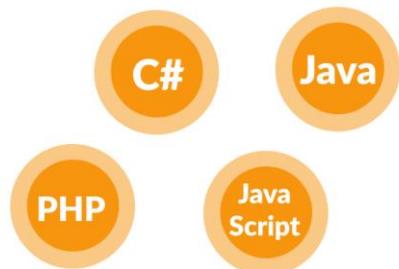
Възможно е не всичко да тръгне гладко от първия път. Примерите по-горе съдържат **неизучаван материал** и имат за цел да ви запалят любопитството и да ви накарат да се поразоровите в Интернет и **да потърсите сами решения** за възникващите трудности.

Ако имате проблеми със задачите по-горе, питайте във **форума на СофтУни**:  
<https://softuni.bg/forum>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 2.2. Прости пресмятания с числа – изпитни задачи

В предходната глава се запознахме със системната конзола и как да работим с нея – как да **прочетем число** от конзолата и как да **отпечатаме резултат** на конзолата. Разгледахме основните аритметични операции и накратко споменахме типовете данни. В настоящата глава ще упражним и затвърдим наученото досега, като разгледаме няколко **по-сложни задачи**, давани на изпити.

## Четене на числа от конзолата

Преди да преминем към задачите, да си припомним най-важното от изучавания материал от предходната тема. Ще започнем с четенето на числа от конзолата.

### Четене на цяло число

Необходима ни е променлива, в която да запазим числото (напр. **num**), и да използваме стандартната команда за четене на данни от конзолата (функцията **input(...)**) в съчетание с функцията **int(...)**, която конвертира текст в число:

```
num = int(input())
```

### Четене на дробно число

По същия начин, както четем цяло число, но този път ще използваме функцията **float(...)**:

```
num = float(input())
```

## Извеждане на текст по шаблон (placeholder)

**Placeholder** представлява израз, който ще бъде заменен с конкретна стойност при отпечатване. Функцията **print(...)** поддържа печatanе на текст по шаблон, като в предходната глава разгледахме няколко възможни начина:

- При първия начин първият аргумент, който трябва да подадем, е форматиращият низ, следван от знака **%** и броя аргументи, равен на броя на плейсхолдърите, изброени в скоби:

```
print("You are %s %s, a %d-years old person from %s." % ("Ivan",  
"Ivanov", "16", "Burgas"))
```

- Втория начин са т.нар "f-strings", при които **пред** форматиращия низ трябва да долепим символа **f**, а аргументите изброяваме в самия низ, между къдрави скоби **{}**:

```
first_name = "Ivan"
last_name = "Ivanov"
age = 21
town = "Burgas"
print(f"You are {first_name} {last_name}, a {age}-years old
person from {town}.")
```

Съществуват и други начини за форматиране на низ, за които по-любознателните от вас могат да прочетат в следната статия, както и да открият сравнения между тях: <https://cito.github.io/blog/f-strings/>.

## Аритметични оператори

Да си припомним основните аритметични оператори за пресмятания с числа.

### Оператор +

```
result = 3 + 5 # резултатът е 8
```

### Оператор -

```
result = 3 - 5 # резултатът е -2
```

### Оператор \*

```
result = 3 * 5 # резултатът е 15
```

### Оператори / и //

```
result = 5 / 2 # резултатът е 2.5 (дробно деление)
result2 = 7 // 3 # резултатът е 2 (целочислено деление)
```

## Конкатенация

При използване на оператора **+** между променливи от тип текст (или между текст и число) се извършва т.нар. конкатенация (слепване на низове):

```
firstName = "Ivan";
lastName = "Ivanov";
age = 19;
str = firstName + " " + lastName + " is " + age + " years old";
# Ivan Ivanov is 19 years old
```

## Изпитни задачи

Сега, след като си припомнихме как се извършват пресмятания с числа и как се четат и печатат числа на конзолата, да минем към задачите. Ще решим няколко задачи от приемен изпит за кандидатстване в СофтУни.

### Задача: учебна зала

Учебна зала има правоъгълен размер  $l$  на  $w$  метра, без колони във вътрешността си. Залата е разделена на две части – лява и дясна, с коридор – приблизително по средата. В лявата и в дясната част има **редици с бюра**. В задната част на залата има голяма **входна врата**. В предната част на залата има **катедра** с подиум за преподавателя. Едно **работно място** заема 70 на 120 см (маса с размер 70 на 40 см + място за стол и преминаване с размер 70 на 80 см). **Коридорът** е широк поне 100 см. Изчислено е, че заради **входната врата** (която е с отвор 160 см) се губи точно 1 работно място, а заради **катедрата** (която е с размер 160 на 120 см) се губят точно 2 работни места. Напишете програма, която въвежда размери на учебната зала и изчислява **броя работни места в нея** при описаното разположение (вж. фигурата).

#### Входни данни

От конзолата се четат 2 числа, по едно на ред:  $l$  (дължина в метри) и  $w$  (широкина в метри).

Ограничения:  $3 \leq w \leq l \leq 100$ .

#### Изходни данни

Да се отпечата на конзолата едно цяло число: **броят места** в учебната зала.

#### Примерен вход и изход

Вход	Изход	Чертеж
8.4 5.2	39	<p>The diagram shows a rectangular room with a central vertical corridor labeled "коридор: 1 м". On the left side, there are several rows of blue rectangles representing desks, with a double door at the bottom labeled "врата". On the right side, there is a yellow rectangular area representing the teacher's desk labeled "катедра". The overall width of the room is indicated as 8.4 and the depth as 5.2.</p>

Вход	Изход	Чертеж
15 8.9	129	<p>коридор: поне 1 м</p> <p>врата</p> <p>катедра</p>

### Пояснения към примерите

В първия пример залата е дълга 1500 см. В нея могат да бъдат разположени **12 реда** ( $12 * 120 \text{ cm} = 1440 + 60 \text{ см остатък}$ ). Залата е широка 890 см. От тях 100 см отиват за коридора в средата. В останалите 790 см могат да се разположат по **11 бюра на ред** ( $11 * 70 \text{ cm} = 770 \text{ cm} + 20 \text{ см остатък}$ ). **Брой места = 12 \* 11 - 3 = 132 - 3 = 129**(имаме 12 реда по 11 места = 132 минус 3 места за катедра и входна врата).

Във втория пример залата е дълга 840 см. В нея могат да бъдат разположени **7 реда** ( $7 * 120 \text{ cm} = 840$ , без остатък). Залата е широка 520 см. От тях 100 см отиват за коридора в средата. В останалите 420 см могат да се разположат по **6 бюра на ред** ( $6 * 70 \text{ cm} = 420 \text{ см}$ , без остатък). **Брой места = 7 \* 6 - 3 = 42 - 3 = 39** (имаме 7 реда по 6 места = 42 минус 3 места за катедра и входна врата).

### Насоки и подсказки

Опитайте първо сами да решите задачата. Ако не успеете, разгледайте насоките и подсказките.

### Идея за решение

Както при всяка една задача по програмиране, е **важно да си изградим идея за решението ѝ**, преди да започнем да пишем код. Да разгледаме внимателно зададеното ни условие. Изисква се да напишем програма, която да изчислява броя работни места в една зала, като този брой е зависим от дължината и височината ѝ. Забелязваме, че те ще ни бъдат подадени като входни данни **в метри**, а информацията за това колко пространство заемат работните места и коридорът, ни е дадена **в сантиметри**. За да извършим изчисленията, ще трябва да използваме еднакви мерни единици, няма значение дали ще изберем да превърнем височината и дължината в сантиметри, или останалите данни в метри. За представеното тук решение е избрана първата опция.

Следва да изчислим **колко колони и колко редици** с бюра ще се съберат. Колоните можем да пресметнем като **от широчината извадим необходимото място за коридора (100 cm)** и **разделим остатъка на 70 cm** (колкото е дължината на едно работно място). Редиците ще намерим като разделим **дължината на 120 cm**. И при двете операции може да се получи **реално число** с цяла и дробна част, в променлива трябва да запазим обаче **само цялата част**. Накрая умножаваме броя на редиците по този на колоните и от него изваждаме 3 (местата, които се губят заради входната врата и катедрата). Така ще получим исканата стойност.

## Избор на типове данни

От примерните входни данни виждаме, че за вход може да ни бъде подадено реално число с цяла и дробна част, затова не е подходящо да избираме тип **int**, нека за тях използваме **float**. Изборът на тип за следващите променливи зависи от метода за решение, който изберем. Както всяка задача по програмиране, тази също има **повече от един начин на решение**.

## Решение

Време е да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

Първото, което трябва да направим, е да прочетем входните данни от конзолата. С функцията **input(...)** четем стойностите от конзолата, а с функцията **float(...)** преобразуваме зададената стрингова (текстова) стойност в **float**:

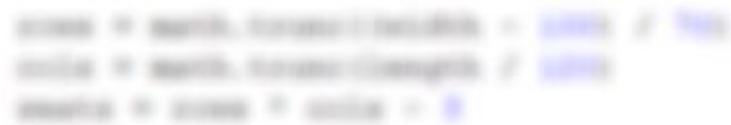
```
length = float(input()) * 100
width = float(input()) * 100
```

Нека пристъпим към изчисленията. Особеното тук е, че след като извършим делението, трябва да запазим в променлива само цялата част от резултата.



**Търсете в Google!** Винаги, когато имаме идея как да решим даден проблем, но не знаем как да го изпишем на Python, или когато се сблъскаме с такъв, за който предполагаме, че много други хора са имали, най-лесно е да се справим като потърсим информация в Интернет.

В случая може да пробваме със следното търсене: "[Python get whole number part of float](#)". Откриваме, че едната възможност е да използваме метода **math.trunc(...)**, като не забравяме да си реферираме библиотеката **math**. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:



Накрая, с функцията **print(...)** отпечатваме резултата на конзолата:

```
print(seats)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1048#0>.

## Задача: зеленчукова борса

Градинар продава реколтата от градината си на зеленчуковата борса. Продава зеленчуци за  $N$  лева на килограм и плодове за  $M$  лева за килограм. Напишете програма, която да пресмята приходите от реколтата в евро (ако приемем, че **едно евро** е равно на **1.94 лв.**).

### Входни данни

От конзолата се четат **4 числа**, по едно на ред:

- Първи ред – Цена за килограм зеленчуци – число с плаваща запетая.
- Втори ред – Цена за килограм плодове – число с плаваща запетая.
- Трети ред – Общо килограми на зеленчуците – цяло число.
- Четвърти ред – Общо килограми на плодовете – цяло число.

Ограничения: Всички числа ще са в интервала от 0.00 до 1000.00

### Изходни данни

Да се отпечата на конзолата **едно число с плаваща запетая**: приходите от всички плодове и зеленчуци в евро.

### Примерен вход и изход

Вход	Изход	Вход	Изход
0.194		1.5	
19.4		2.5	
10	101	10	20.6185567010309
10		10	

Пояснения към първия пример:

- Зеленчуците струват: 0.194 лв. \* 10 кг. = **1.94 лв.**
- Плодовете струват: 19.4 лв. \* 10 кг. = **194 лв.**

- Общо: 195.94 лв. = 101 евро.

## Насоки и подсказки

Първо ще дадем няколко разсъждения, а след това и конкретни насоки за решаване на задачата, както и съществената част от кода.

### Идея за решение

Нека първо разгледаме зададеното ни условие. В случая, от нас се иска да пресметнем колко е **общият приход** от реколтата. Той е равен на **сбора от печалбата от плодовете и зеленчуците**, а тях можем да изчислим като умножим цената на килограм по количеството им. Входните данни са дадени в лева, а за изхода се изисква да бъде в евро. По условие 1 евро е равно на 1.94 лева, следователно, за да получим исканата **изходна стойност**, трябва да разделим **сбора на 1.94**.

### Избор на типове данни

След като сме изяснили идеята си за решаването на задачата, можем да пристъпим към избора на подходящи типове данни. Да разгледаме **входа**: дадени са **две цели числа** за общия брой килограми на зеленчуците и плодовете, съответно променливите, които декларираме, за да пазим техните стойности, ще бъдат от тип **int**. За цените на плодовете и зеленчуците е указано, че ще бъдат подадени **две числа с плаваща запетая**, т.е. променливите ще бъдат от тип **float**.

Може да декларираме също две помощни променливи, в които да пазим стойността на печалбата от плодовете и зеленчуците поотделно. Като **изход** се изиска **число с плаваща запетая**, т.е. резултата трябва да е от тип **float**.

### Решение

Време е да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

За да прочетем входните данни, декларираме променливи, като внимаваме да ги именуваме по такъв начин, който да ни подсказва какви стойности съдържат променливите. С методите **parseInt(...)** и **parseFloat(...)** преобразуваме зададената текстова стойност съответно в цяло и дробно число.

За да прочетем входните данни декларираме променливи, като внимаваме да ги именуваме по такъв начин, който да ни подсказва какви стойности съдържат променливите. С **input(...)** четем стойностите от конзолата, а с функциите **int(...)** и **float(...)** преобразуваме зададената стрингова стойност съответно в цяло и дробно число:

```
vegetables_price = float(input())
fruits_price = float(input())
```

След което, извършваме необходимите изчисления:

```
vegetables_total_price = vegetables_price * vegetables_quantity
fruits_total_price = fruits_price * fruits_quantity
```

В условието на задачата не е зададено специално форматиране на изхода, следователно трябва просто да изчислим исканата стойност и да я отпечатаме на конзолата. Както в математиката, така и в програмирането делението има приоритет пред събирането. За задачата обаче трябва първо да **сметнем сбора** на двете получени стойности и след това да **разделим на 1.94**. За да дадем предимство на събирането, може да използваме скоби. С **print(...)** отпечатваме изхода на конзолата:

```
print((vegetables_total_price + fruits_total_price) / 1.94)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1048#1>.

## Задача: ремонт на плочки

На площадката пред жилищен блок трябва да се поставят плочки. Площадката е с форма **на квадрат със страна N метра**. Плочките са широки „W“ метра и дълги „L“ метра. На площадката има една пейка с **ширина M метра** и **дължина O метра**. Под нея не е нужно да се слагат плочки. Всяка плочка се поставя за **0.2 минути**.

Напишете програма, която чете от конзолата размерите на **площадката, плочките и пейката** и пресмята **колко плочки са необходими да се покрие площадката и пресмята времето за поставяне на всички плочки**.

Пример: площадка с размер 20 м. има площ 400 кв.м. Пейка, широка 1 м. и дълга 2 м., заема площ 2 кв.м. Една плочка е широка 5 м. и дълга 4 м. и има площ = 20 кв.м. Площта, която трябва да се покрие, е  $400 - 2 = 398$  кв.м. Необходими са  $398 / 20 = 19.90$  плочки. Необходимото време е  $19.90 * 0.2 = 3.98$  минути.

## Входни данни

От конзолата се четат **5 числа**:

- N – дължината на страна от **площадката** в интервала [1 ... 100].
- W – широчината на една **плочка** в интервала [0.1 ... 10.00].
- L – дължината на една **плочка** в интервала [0.1 ... 10.00].
- M – широчината на **пейката** в интервала [0 ... 10].
- O – дължината на **пейката** в интервала [0 ... 10].

## Изходни данни

Да се отпечатат на конзолата **две числа**:

- броя **плочки**, необходим за ремонта
- времето за поставяне

Всяко число да бъде на нов ред и закръглено до втория знак след десетичната запетая.

## Примерен вход и изход

Вход	Изход	Вход	Изход
20		40	
5		0.8	
4	19.9	0.6	3302.08
1		3	660.42
2		5	

Обяснение към примера:

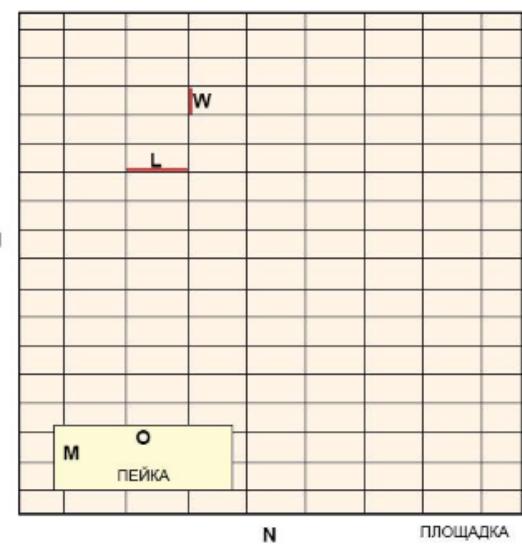
- Обща площ =  $20 * 20 = 400$ .
- Площ на пейката =  $1 * 2 = 2$ .
- Площ за покриване =  $400 - 2 = 398$ .
- Площ на плочки =  $5 * 4 = 20$ .
- Необходими плочки =  $398 / 20 = 19.9$ .
- Необходимо време =  $19.9 * 0.2 = 3.98$ .

## Насоки и подсказки

Нека да си направим чертеж, за да поясним условието на задачата. Той може да изглежда по начина, показан на картинката.

### Идея за решение

Изиска се да пресметнем **броя плочки**, който трябва да се постави, както и времето, за което това ще се извърши. За да **изчислим броя**, е необходимо да сметнем **площта**, която трябва да се покрие, и да я **разделим на лицето на една плочка**. По условие площадката е квадратна, следователно общата площ ще намерим, като умножим страната ѝ по стойността ѝ  $N * N$ . След това пресмятаме **площта**, която заема пейката, също



като умножим двете ѝ страни **M \* O**. Като извадим площта на пейката от тази на цялата площадка, получаваме площта, която трябва да се ремонтира.

Лицето на единична плочка изчисляваме като **умножим едната ѝ страна по другата W \* L**. Както вече отбелязахме, сега трябва да **разделим площта за покриване на площта на една плочка**. По този начин ще разберем какъв е необходимият брой плочки. Него умножаваме по **0.2** (времето, за което по условие се поставя една плочка). Така вече ще имаме исканите изходни стойности.

## Избор на типове данни

Дължината на страна от площадката, широчината и дължината на пейката ще бъдат дадени като **цили числа**, следователно, за да преобразуваме техните стойности може да използваме системната функция **int(...)**. За широчината и дължината на плочките ще ни бъдат подадени реални числа (с цяла и дробна част), затова за тях ще използваме **float(...)**.

## Решение

Както и в предходните задачи, можем мислено да разделим решението на части:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

Първото, което трябва да направим, е да разгледаме **входните данни** на задачата. Важно е да внимаваме за последователността, в която са дадени. С **input(...)** четем стойностите от конзолата, а с **int(...)** и **float(...)** преобразуваме зададената стрингова стойност, съответно в **int** и **float**:

```
# Ground length
n = int(input())
# Tile width
w = float(input())
# Tile length
l = float(input())
# Bench width
m = int(input())
# Bench length
o = int(input())
```

След като сме инициализирали променливите и сме запазили съответните стойности в тях, пристъпваме към **изчисленията**. Кодът по-долу е нарочно даден замъглен, за да може читателят да помисли самостоятелно над него:

```
area_with_tiles = float(input())
tile_area = float(input())
time = area_with_tiles / tile_area * 0.2
```

Накрая **изчисляваме стойностите**, които трябва да отпечатаме на конзолата. **Броят** на необходимите **плочки** получаваме, като **разделим площта**, която трябва да се покрие, **на площта на единична плочка**:

```
tiles_count = area_with_tiles / tile_area
time = tiles_count * 0.2
```

В условието на задачата е зададено закръгление на изхода **до втория знак след десетичната запетая**. Затова не можем просто да отпечатаме стойностите с **print(...)**. Ще използваме функцията **round(...)**, която закръгля подаденото число до най-близкото число, със точност n-цифри след десетичната запетая. Например числото 1.35 ще се закръгли до 1, а 1.65 - до 2:

```
print(round(tiles_count, 2))
print(round(time, 2))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1048#2>.

## Задача: парички

Преди време Пешо си е купил биткойни. Сега ходи на екскурзия из Европа и ще му трябва евро. Освен биткойни има и китайски юани. Пешо иска да обмени парите си в евро за екскурзията. Напишете програма, която да пресмята колко евро може да купи спрямо следните валутни курсове:

- 1 биткойн = 1168 лева
- 1 доллар = 1.76 лева
- 1 китайски юан = 0.15 долара
- 1 евро = 1.95 лева

Обменното бюро има комисионна от 0% до 5% от крайната сума в евро.

### Входни данни

От конзолата се четат 3 числа:

- На първия ред – **броят биткойни**. Цяло число в интервала [0 ... 20].
- На втория ред – **броят китайски юани**. Реално число в интервала [0.00 ... 50 000.00].
- На третия ред – **комисионната**. Реално число в интервала [0.00 ... 5.00].

### Изходни данни

На конзолата да се отпечата 1 число - резултатът от обмяната на валутите. Резултатът да се форматира до втората цифра след десетичния знак.

## Примерен вход и изход

Вход	Изход
1	
5	569.67
5	

Вход	Изход
20	
5678	12442.24
2.4	

Вход	Изход
7	
50200.12	10659.47
3	

Обяснение:

- 1 биткойн = 1168 лева
- 5 юана = 0.75 долара
- 0.75 долара = 1.32 лева
- $1168 + 1.32 = 1169.32$  лева = 599.651282051282 евро
- Комисионна: 5% от 599.651282051282 = 29.9825641025641
- Резултат:  $599.651282051282 - 29.9825641025641 = 569.668717948718 = 569.67$  евро

## Насоки и подсказки

Нека отново помислим първо за начина, по който можем да решим задачата, преди да започнем да пишем код.

### Идея за решение

Виждаме, че ще ни бъдат подадени броят биткойни и броят китайски юани. За изходната стойност е указано да бъде в евро. В условието са посочени и валутните курсове, с които трябва да работим. Забелязваме, че към евро можем да преобразуваме само сума в лева, следователно трябва **първо да пресметнем цялата сума**, която Пешо притежава в лева, и след това да **изчислим изходната стойност**.

Тъй като ни е дадена информация за валутния курс на биткойни срещу лева, можем директно да направим това преобразуване. От друга страна, за да получим стойността на **китайските юани в лева**, трябва първо да ги **конвертираме в долари**, а след това **доларите - в лева**. Накрая ще **съберем двете получени стойности** и ще пресметнем на колко евро съответстват.

Остава последната стъпка: да **пресметнем колко ще бъде комисионната** и да **извадим получената сума от общата**. Като комисионна ще ни бъде подадено **реално число**, което ще представлява определен **процент от общата сума**. Нека още в началото разделим подаденото число на 100, за да изчислим **процентната му стойност**. Няя ще умножим по сумата в евро, а резултатът ще извадим от същата тази сума. Получената сума ще отпечатаме на конзолата.

## Избор на типове данни

Биткойните са дадени като **цяло число**, следователно за тяхната стойност може да използваме **променлива от тип `int`**. Като брой китайски юани и комисационна ще получим **реално число**, следователно за тях използваме **`double`**. Тъй като **`double`** е типът данни с по-голям обхват, а **изходът** също ще бъде **реално число**, ще използваме него и за останалите променливи, които създаваме.

Биткойните са дадени като **цяло число**, следователно за тяхната стойност може да декларираме променлива преобразувана с функцията **`int(...)`**. Като брой китайски юани и комисационна ще получим **реално число**, следователно за тях ще използваме **`float(...)`**.

## Решение

След като сме си изградили идея за решението на задачата и сме избрали структурите от данни, с които ще работим, е време да пристъпим към **писането на код**. Както и в предните задачи, можем да разделим решението на три подзадачи:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

Декларираме променливите, които ще използваме, като отново внимаваме да изберем **смислени имена**, които подсказват какви данни съдържат те. Инициализираме техните стойности: създаваме си променливи, в който да запазим подадените на функцията стрингови аргументи, като ги конвертираме към цяло или дробно число:

```
bitcoins = int(input())
yuans = float(input())
commission = float(input()) / 100
```

Извършваме необходимите изчисления:

```
bitcoins_to_leva = bitcoins * 1168
yuans_to_dollars = yuans * 0.15
dollars_to_leva = yuans_to_dollars * 1.76
```

Накрая пресмятаме стойността на комисационната и я изваждаме от сумата в евро. Нека обърнем внимание на начина, по който можем да изпишем това: **`euro -= commission * euro`** е съкратен начин за изписване на **`euro = euro - (commission * euro)`**. В случая използваме **комбиниран оператор за присвояване** (**`=`**), който изважда стойността от операнда вдясно от този вляво. Операторът за

умножение (\*) има по-висок приоритет от комбинирания оператор, затова изразът **commission \* euro** се изпълнява първи, след което неговата стойност се изважда.

Накрая остава да изведем резултата на конзолата. Забелязваме, че се изисква форматиране на числената стойност до втория знак след десетичната точка. За разлика от предходната задача, тук дори и числото да е цяло, **трябва винаги да има два знака след десетичната точка** (например **5.00**). Един вариант е да използваме функцията **print(...)**, като форматираме стринг по зададен шаблон (**.2f**). Чрез този подход можем да преобразуваме числото в текст, запазвайки определен брой знаци след десетичната запетая:

```
euro -= euro * commission
print("%.2f" % euro)
```

Нека обърнем внимание на нещо, което важи за всички задачи от този тип: разписано по този начин, решението на задачата е доста подробно. Тъй като условието като цяло не е сложно, бихме могли на теория да напишем един голям израз, в който директно след получаване на входните данни да сметнем изходната стойност. Например такъв израз би изглеждал ето така:

```
euro = ((bitcoins * 1168) + (yuans * 0.15 * 1.76)) / 1.95
      - (commission * ((bitcoins * 1168)
      + (yuans * 0.15 * 1.76)) / 1.95)
```

Този код би дал правилен резултат, но се чете трудно. Няма да ни е лесно да разберем какво прави, дали съдържа грешки и ако има такива - как да ги поправим. По-добра практика е **вместо един сложен израз да напишем няколко прости** и да запишем резултатите от тях в променливи със подходящи имена. Така кодът е ясен, по-лесно четим и променяме.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1048#3>.

## Задача: дневна печалба

Иван е програмист в американска компания и **работи** от вкъщи **средно N дни в месеца**, като изкарва **средно по M долара на ден**. В края на годината Иван **получава бонус**, който е **равен на 2.5 месечни заплати**. От спечеленото през годината му се **удържат 25% данъци**. Напишете програма, която да **пресмята колко е чистата средна печалба** на Иван на ден в лева, тъй като той харчи изкараното в България. Приема се, че в годината има точно 365 дни. Курсът на долара спрямо лева ще се подава на функцията.

## Входни данни

От конзолата се четат 3 числа:

- На първия ред – **работни дни в месеца**. Цяло число в интервала [5 ... 30].
- На втория ред – **изкарани пари на ден**. Реално число [10.00 ... 2000.00].
- На третия ред – **курсът на долара спрямо лева /1 доллар = X лева/**. Реално число в интервала [0.99 ... 1.99].

## Изходни данни

На конзолата да се отпечата едно число – средната печалба на ден в лева. Резултатът да се форматира до втората цифра след десетичния знак.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
21		15		22	
75.00	74.61	105	80.24	199.99	196.63
1.59		1.71		1.50	

Обяснение:

- 1 месечна заплата =  $21 * 75 = 1575$  долара.
- Годишен доход =  $1575 * 12 + 1575 * 2.5 = 22837.5$  долара.
- Данък = 25% от 22837.5 = 5709.375 лева.
- Чист годишен доход =  $17128.125$  долара = 27233.71875 лева.
- Средна печалба на ден =  $27233.71875 / 365 = 74.61$  лева.

## Насоки и подсказки

Първо да анализираме задачата и да измислим как да я решим. След това ще изберем типовете данни и накрая ще напишем кода на решението.

## Идея за решение

Нека първо пресметнем колко е месечната заплата на Иван. Ще умножим работните дни в месеца по парите, които той печели на ден. Умножаваме получения резултат първо по 12, за да изчислим колко е заплатата му за 12 месеца, а след това и по 2.5, за да пресметнем бонуса. Като съберем двете получени стойности, ще изчислим общия му годишен доход. От него трябва да извадим 25%. Това може да направим като умножим общия доход по 0.25 и извадим резултата от него. Спрямо дадения ни курс преобразуваме долларите в лева, след което разделяме резултата на дните в годината (приемаме, че са 365).

## Избор на типове данни

Работните дни за месец са дадени като **цяло число**, следователно за тяхната стойност може да декларираме променлива, в която да конвертираме подадената

стрингова стойност до число с функцията **int(...)**. За изкараните пари, както и за курса на долара спрямо лева, ще ни бъдат подадени реални числа, следователно за тях ще използваме функцията **float(...)**.

## Решение

Отново, след като имаме идея как да решим задачата и сме помислили за типовете данни, с които ще работим, пристъпваме към **писането на програмата**. Както и в предходните задачи, можем да разделим решението на три подзадачи:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

Декларираме променливите, които ще използваме, като отново се стараем да изберем **подходящи имена**. Създаваме си променливи, в които запазваме прочетените стойности от конзолата, като преобразуваме стринга към цяло или дробно число с **int(...)** / **float(...)**:

Извършваме изчисленията:

```
monthly_salary = working_days * profit_per_day
annual_earnings = monthly_salary * 12 + monthly_salary * 2.5
taxes = annual_earnings * 0.25
net_annual_earnings = (annual_earnings - taxes)
salary_in_leva = net_annual_earnings * currency_rate
```

Бихме могли да напишем израза, с който пресмятаме общия годишен доход, и без скоби. Тъй като умножението е операция с по-висок приоритет от събирането, то ще се извърши първо. Въпреки това **писането на скоби** се препоръчва, когато използваме повече оператори, защото така кодът става по-лесно четим и възможността да се допусне грешка е по-малка.

Накрая остава да изведем резултата на екрана. Забелязваме, че се **изисква форматиране на числената стойност до втория знак след десетичната точка**. Можем да използваме същия подход от предходната задача: форматиране стринг по зададен шаблон:

```
print("%.2f" % (salary_in_leva / 365))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1048#4>.

# Глава 3.1. Прости проверки

В настоящата глава ще разгледаме условните конструкции в езика Python, чрез които нашата програма може да има различно поведение, в зависимост от дадено условие. Ще обясним синтаксиса на условните оператори за проверки (**if**, **if-elif** и **else**) с подходящи примери и ще видим в какъв диапазон живее една променлива (нейният **обхват**). Накрая ще разгледаме техники за **дебъгване**, чрез които постъпково да проследяваме пътя, който извървява нашата програма по време на своето изпълнение.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=cQII0wQLVRE>.

## Сравняване на числа

В програмирането можем да сравняваме стойности чрез следните **оператори**:

- Оператор **<** (по-малко)
- Оператор **>** (по-голямо)
- Оператор **<=** (по-малко или равно)
- Оператор **>=** (по-голямо или равно)
- Оператор **==** (равно)
- Оператор **!=** (различно)

При сравнение резултатът е буlevа стойност – **True** или **False**, в зависимост от това дали резултатът от сравнението е истина или лъжа.

## Примери за сравнение на числа

```
a = 5
b = 10
print(a > b)          # False
print(a == b)          # False
print(a < 10)          # True
print(b < 10)          # False
print(a != b)          # True
print(b >= 10)          # True
print(b == a * 2)      # True
```

Обърнете внимание, че при отпечатване на стойностите **true** и **false** в езика Python, те се отпечатват с главна буква, съответно **True** и **False**.

## Оператори за сравнение

В езика Python можем да използваме следните оператори за сравнение на данни:

Оператор	Означение	Работи за
Проверка за равенство	<code>==</code>	
Проверка за различие	<code>!=</code>	числа, стрингове, дати
По-голямо	<code>&gt;</code>	
По-голямо или равно	<code>&gt;=</code>	числа, дати, други сравними типове
По-малко	<code>&lt;</code>	
По-малко или равно	<code>&lt;=</code>	

Ето един пример:

```
result = "hello" == "Hello"
print(result) # False
```

## Прости проверки

В програмирането често **проверяваме дадени условия** и извършваме различни действия, спрямо резултата от тези проверки. Проверките извършваме посредством **if** клаузи, които имат следната конструкция:

```
if условие:
    # тяло на условната конструкция
```

## Пример: отлична оценка

Въвеждаме оценка в конзолата и проверяваме дали тя е отлична (**≥ 5.50**).

```
grade = float(input())
if grade >= 5.5:
    print("Excellent!")
```

Тествайте кода от примера локално. Опитайте да въведете различни оценки, например 4.75, 5.49, 5.50 и 6.00. При оценки по-малки от 5.50 програмата няма да изведе нищо, а при оценка 5.50 или по-голяма, ще изведе "Excellent!".

## Тестване в Judge системата

Тествайте програмата от примера в Judge системата на СофтУни:  
<https://judge.softuni.bg/Contests/Practice/Index/1049#0>.

## Проверки с if-else конструкция

Конструкцията **if** може да съдържа и **else** клауза, с която да окажем конкретно действие в случай, че булевият израз (който е зададен в началото **if булев израз**) върне отрицателен резултат (**False**). Така построена, **условната конструкция** наричаме **if-else** и поведението ѝ е следното: ако резултатът от условието е **позитивен (True)** - извършваме едни действия, а когато е **негативен (False)** - други. Форматът на конструкцията е:

```
if условие:  
    # тяло на условната конструкция  
else:  
    # тяло на else конструкция
```

### Пример: отлична оценка или не

Подобно на горния пример, въвеждаме оценка, проверяваме дали е отлична, но изписваме резултат и в двата случая.

```
grade = float(input())  
if grade >= 5.5:  
    print("Excellent!")  
else:  
    print("Not excellent.")
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#1>.

## За блоковете от код

Чрез **табулация** създаваме блок от код, чрез който може да се изпълняват група от команди. Когато имаме код в **if, elif, else** (и други конструкции) искаме да изпълним поредица от операции, ги поставяме в блок след условието.



Добра практика е, да ползваме табулация (или четири spaces), понеже това прави кода ни по-четим и по-подреден и по този начин избягваме грешки по време на изпълнение на кода.

Ето един пример за лоша индентация:

```
firstNum = 4  
secondNum = 1  
if firstNum > secondNum:
```

```

result = firstNum - secondNum
print(result)
elif secondNum > firstNum:
    result = secondNum - firstNum
print(result)

```

Горният код или ще даде грешка, защото е грешно форматиран, или изпълнението му ще изведе **грешен резултат** на конзолата:

The screenshot shows the CodeBlocks IDE interface. In the terminal window, there are several errors displayed:

- A red arrow icon indicates an error.
- The text "F:\Desktop\PythonBook\venv\Scripts\python.exe" is shown.
- Three "3" characters are printed, which are likely artifacts from the invalid code execution.
- The message "Process finished with exit code 0" is at the bottom.

С правилна индентация:

```

firstNum = 4
secondNum = 1
result = 0
if firstNum > secondNum:
    result = firstNum - secondNum
elif secondNum > firstNum:
    result = secondNum - firstNum
print(result)

```

На конзолата ще бъде отпечатано следното:

The screenshot shows the CodeBlocks IDE interface. In the terminal window, the output is correct:

- A green arrow icon indicates success.
- The text "F:\Desktop\PythonBook\venv\Scripts\python.exe" is shown.
- The number "3" is printed.
- The message "Process finished with exit code 0" is at the bottom.

## Пример: четно или нечетно

Да се напише програма, която проверява, дали дадено цяло число е **четно** (even) или **нечетно** (odd).

Задачата можем да решим с помощта на една **if-else** конструкция и оператора **%**, който връща **остатък при деление** на две числа:

```
number = int(input())
if number % 2 == 0:
    print("even")
else:
    print("odd")
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#2>.

## Пример: по-голямото число

Да се напише програма, която чете две цели числа и извежда по-голямото от тях.

Първата ни подзадача е да прочетем двете числа. След което, чрез приста **if-else** конструкция, в съчетание с оператора за по-голямо (**>**), да направим проверка. Кодът е замъглен умишлено и трябва да бъде довършен от читателя:

```
print("Enter two integers: ")
firstNumber = int(input())
secondNumber = int(input())
```



## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#3>.

## Живот на променлива

Всяка една променлива си има обхват, в който съществува, наречен **variable scope**. Този обхват уточнява къде една променлива може да бъде използвана. В езика Python променливите могат да бъдат използвани навсякъде, стига да са инициализирани поне веднъж.

В примера по-долу, на последния ред, на който се опитваме да отпечатаме променливата **my\_name**, която е дефинирана в **else** конструкцията, ще получим грешка, защото в конкретния случай не се е изпълнило тялото на **else** клаузата, в която инициализираме променливата. Но отпечатването на променливата **can\_drive** е без проблемно, защото програмата е влязла в тялото на **if** клаузата и е инициализирала променливата. Както забелязвате обаче променливите **can\_drive** и **my\_name** за оцветени в жълто. Това е предупреждение от **PyCharm**, че е възможно да получим грешка. Затова е най-добре да внимаваме с това къде инициализираме променливите.

```

my_age = 20
if my_age >= 18:
    can_drive = True
else:
    my_name = 'Ivan'

print(my_age)
print(can_drive)
print(my_name)

```

## Серии от проверки

Понякога се налага да извършим серия от проверки, преди да решим какви действия ще изпълнява нашата програма. В такива случаи, можем да приложим конструкцията **if-elif-...-else в серия**. За целта използваме следния формат:

```

if условие:
    # тяло на условната конструкция
elif условие2:
    # тяло на условната конструкция
elif условие3:
    # тяло на условната конструкция
...
else:
    # тяло на else конструкция

```

## Пример: число от 1 до 9 на английски

Да се изпише число в интервала от 1 до 9 с текст на английски език (числото се чете от конзолата). Можем да прочетем числото и след това чрез **серия от проверки** отпечатваме съответстващата му английска дума:

```

number = int(input())
if number == 1:
    print("one")
elif num == 2:
    print("two")
elif ...:
    ...
elif num == 9:
    print("nine")
else:
    print("number too big")

```

Програмната логика от примера по-горе **последователно сравнява** входното число от конзолата с цифрите от 1 до 9, като **всяко следващо сравнение се извършва**, само в случай че предходното сравнение не е било истина. В крайна сметка, ако никое от **if** условията не е изпълнено, се изпълнява последната **else** клауза.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#4>.

## Упражнения: прости проверки

За да затвърдим знанията си за условните конструкции **if** и **if-elif**, ще решим няколко практически задачи.

### Задача: бонус точки

Дадено е **цяло число** – брой точки. Върху него се начисляват **бонус точки** по правилата, описани по-долу. Да се напише програма, която пресмята **бонус точките** за това число и **общия брой точки** с бонусите.

- Ако числото е **до 100** включително, бонус точките са 5.
- Ако числото е **по-голямо от 100**, бонус точките са **20%** от числото.
- Ако числото е **по-голямо от 1000**, бонус точките са **10%** от числото.
- Допълнителни бонус точки (начисляват се отделно от предходните):
  - За **четно** число -> + 1 т.
  - За число, което завършва на 5 -> + 2 т.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
20	6	175	37	2703	270.3	15875	1589.5
	26		212		2973.3		17464.5

### Насоки и подсказки

Основните и допълнителните бонус точки можем да изчислим с поредица от няколко **if-elif-else** проверки. Като за **основните бонус точки имаме 3 случая** (когато въведеното число е до 100, между 100 и 1000 и по-голямо от 1000), а за **допълнителните бонус точки - още 2 случая** (когато числото е четно и нечетно):

```
points = int(input())
bonus = 0

if points > 1000:
    bonus += points * 0.1
elif #TODO: finish the logic
```

```

elif #TODO: finish the logic
if #TODO: finish the logic
if #TODO: finish the logic

print(bonus)
sum = bonus + points
print(sum)

```

Ето как би могло да изглежда решението на задачата в действие:

```

Run CodeBlocks
F:\Desktop\PythonBook\venv\Scripts\python.exe
F:\Desktop\PythonBook\PythonBook.py
Enter score: 20
Bonus score: 6
Total score: 26
Process finished with exit code 0

```

Обърнете внимание, че за тази задача Judge е настроен да игнорира всичко, което не е число, така че можем да печатаме не само числата, но и уточняващ текст.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#5>.

## Задача: сумиране на секунди

Трима спортни състезатели финишират за някакъв **брой секунди** (между 1 и 50). Да се напише програма, която въвежда времената на състезателите и пресмята **сумарното им време** във формат "минути:секунди". Секундите да се изведат с водеща нула (2 -> "02", 7 -> "07", 35 -> "35").

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
35		22		50		14	
45	2:04	7	1:03	50	2:29	12	0:36
44		34		49		10	

### Насоки и подсказки

Първо сумираме трите числа, за да получим общия резултат в секунди. Понеже **1 минута = 60 секунди**, ще трябва да изчислим броя минути и броя секунди в диапазона от 0 до 59:

- Ако резултатът е между 0 и 59, отпечатваме 0 минути + изчислените секунди.

- Ако резултатът е между 60 и 119, отпечатваме 1 минута + изчислените секунди минус 60.
- Ако резултатът е между 120 и 179, отпечатваме 2 минути + изчислените секунди минус 120.
- Ако секундите са по-малко от 10, извеждаме водеща нула преди тях.

```
firstCompetitor = int(input())
# TODO: Read second and third competitors' time
seconds =
    firstCompetitor + secondCompetitor + thirdCompetitor
minutes = 0

if seconds > 59:
    minutes += 1
    seconds = seconds - 60
if seconds > 59:
    minutes += 1
    seconds = seconds - 60
if seconds < 10:
    print(f'{minutes}:{0{seconds}}')
else:
    print(f'{minutes}:{seconds}')
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#6>.

### Задача: конвертор за мерни единици

Да се напише програма, която преобразува разстояние между следните **8 мерни единици**: **m, mm, cm, mi, in, km, ft, yd**. Използвайте съответствията от таблицата:

Входна единица	Изходна единица
1 meter (m)	1000 millimeters (mm)
1 meter (m)	100 centimeters (cm)
1 meter (m)	0.000621371192 miles (mi)
1 meter (m)	39.3700787 inches (in)
1 meter (m)	0.001 kilometers (km)
1 meter (m)	3.2808399 feet (ft)
1 meter (m)	1.0936133 yards (yd)

Входните данни се състоят от три реда:

- Първи ред: число за преобразуване.
- Втори ред: входна мерна единица.
- Трети ред: изходна мерна единица (за резултата).

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
12 km ft	39370.0788	150 mi in	9503999.99393599		
				450 yd km	0.41147999937455

### Насоки и подсказки

Прочитаме си входните данни, като към прочитането на мерните единици можем да добавим функцията `lower()`, която ще направи всички букви малки. Както виждаме от таблицата в условието, можем да конвертираме само **между метри и някаква друга мерна единица**. Следователно трябва първо да изчислим числото за преобразуване в метри. Затова трябва да направим набор от проверки, за да определим каква е входната мерна единица, а след това и за изходната мерна единица:

```
size = float(input())
sourceMetric = input().lower()
destMetric = input().lower()
if sourceMetric == "km":
    size = size / 0.001
# Check the other metrics: mm, cm, ft, yd, ...
if destMetric == "ft":
    size = size * 3.2808399
# Check the other metrics: mm, cm, ft, yd, ...
print(f'{size} {destMetric}')
```

### Тестване в Judge системата

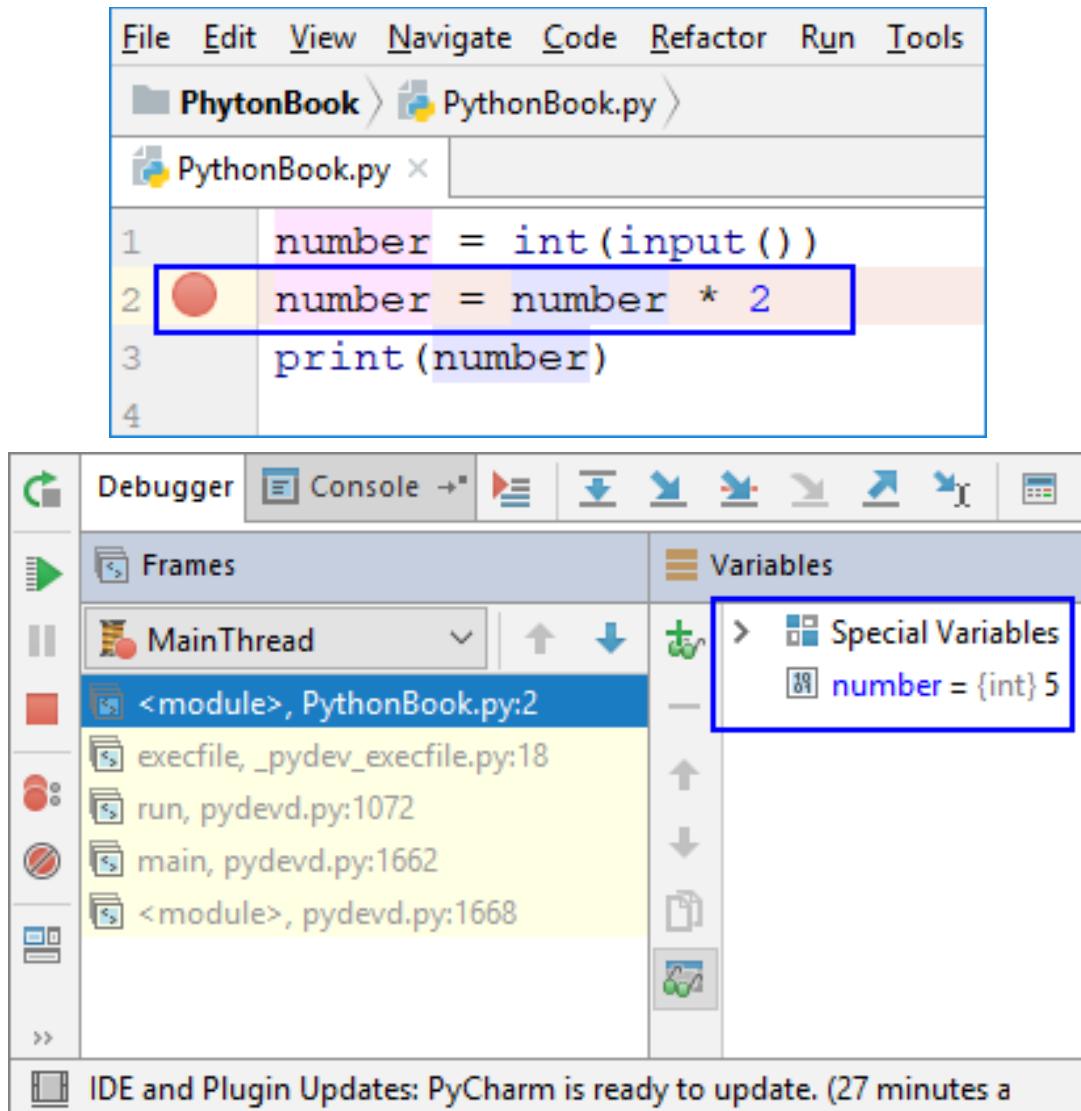
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#7>.

## Дебъгване - прости операции с дебъгер

До момента писахме доста код и често пъти в него имаше грешки, нали? Сега ще покажем един инструмент, с който можем да намираме грешките по-лесно.

### Какво е "дебъгване"?

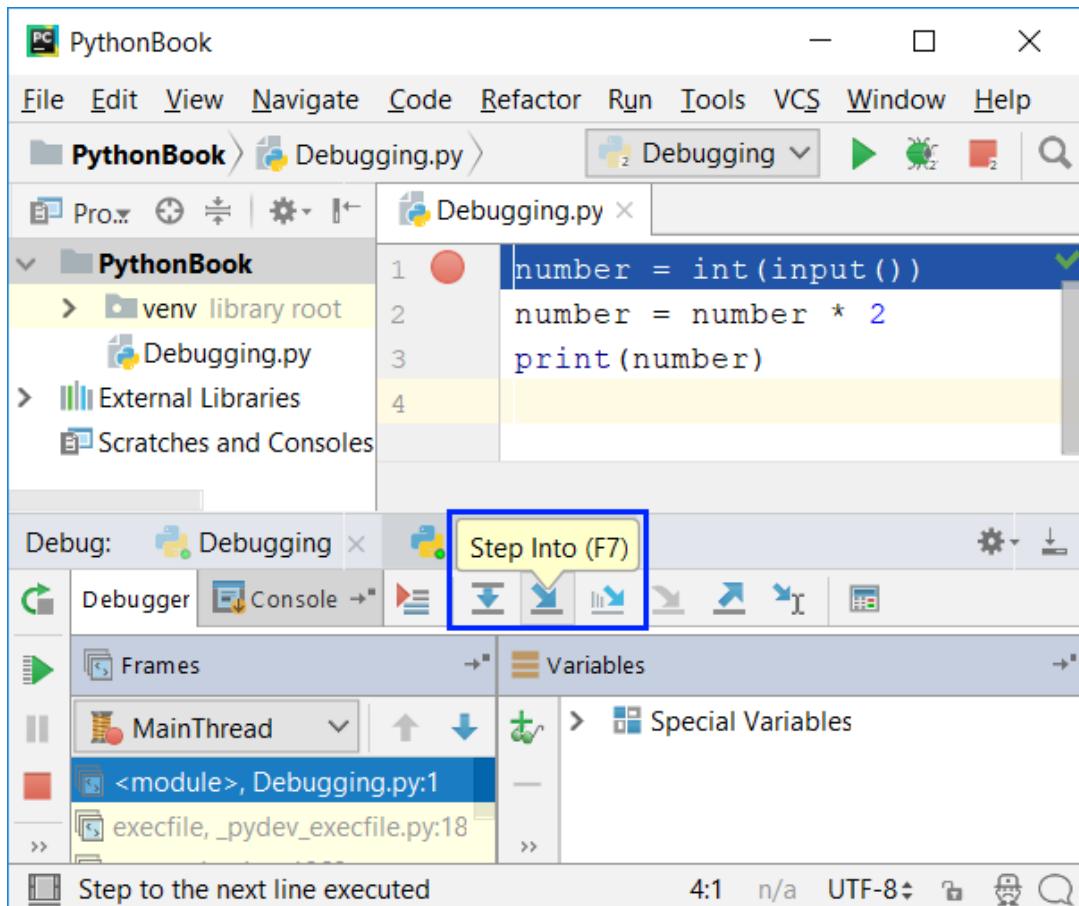
Дебъгване е процесът на "закачане" към изпълнението на програмата, който ни позволява да проследим поетапно процеса на изпълнение. Можем да следим **ред по ред** какво се случва с нашата програма, какъв път следва, какви стойности имат дефинираните променливи на всяка стъпка от изпълнението на програмата и много други неща, които ни позволяват да откриваме грешки (**бъгове**):



## Дебъгване в PyCharm

Чрез натискане на [Shift + F9], стартираме програмата в Debug режим. Преминаваме към **следващия ред** на изпълнение с [F7].

Чрез [Ctrl + F8] създаваме стопери - така наречените **breakpoints**, до които можем да стигнем директно използвайки [Shift + F9] (при стартирането на програмата в Debug режим).



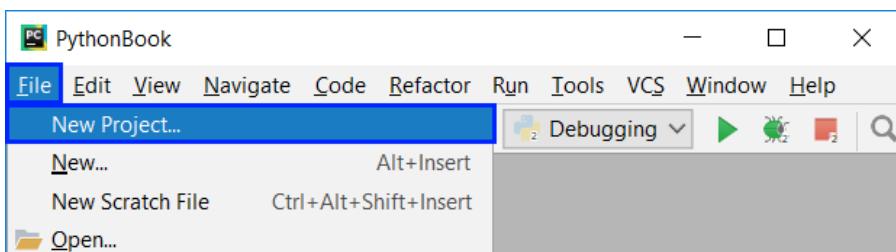
## Упражнения: прости проверки

Нека затвърдим наученото в тази глава с няколко задачи.

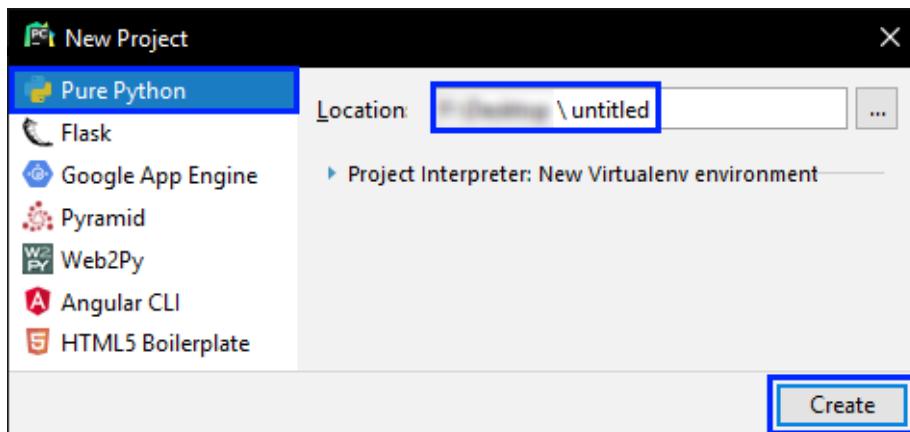
### Празно PyCharm решение (Project)

Създаваме празно решение в **PyCharm**, за да организираме по-добре решенията на задачите от упражненията – всяка задача ще бъде в отделен файл и всички задачи ще бъдат в общ Project.

Стартираме PyCharm. Създаваме нов Project: [File] -> [New Project]:



Избираме от полето в ляво **Pure Python** и задаваме директория на проекта, като на мястото на **untitled** слагаме името на нашия проект:



Сега имаме създаден празен проект (без файлове в него).

### Задача: проверка за отлична оценка

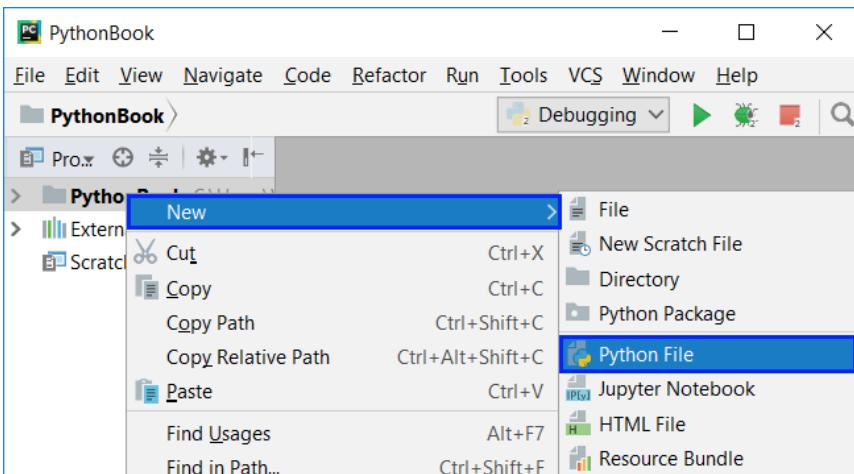
Първата задача от упражненията за тази тема е да се напише **конзолна програма**, която **въвежда оценка**(десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока.

#### Примерен вход и изход

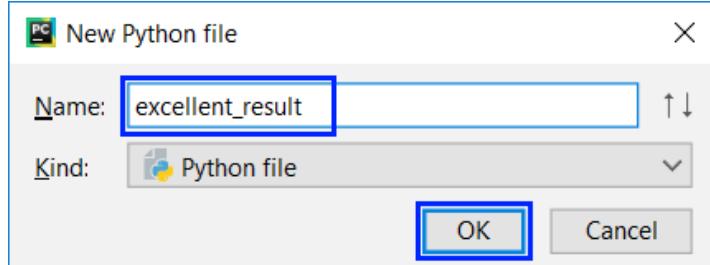
Вход	Изход	Вход	Изход	Вход	Изход
6	Excellent!	5.5	Excellent!	5.49	(няма изход)

#### Насоки и подсказки

Създаваме нов **python** файл (**.py**) като цъкнем с десен клавиш на мишката върху създадената от нас папка и изберем [New] -> [Python File]:



Ще се отвори диалогов прозорец за избор на име на файла. Тъй като задачата ни е за проверка на отлична оценка, нека именуваме файла **excellent\_result**:



Вече имаме Project с един файл в него. Остава да напишем кода за решаване на задачата. За целта пишем следния код:

```

1 grade = float(input())
2 if grade >= 5.5:
3     print("Excellent!")
4

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#0>.

### 01. Excellent Result

```

1 grade = float(input())
2 if grade >= 5.5:
3     print("Excellent!")

```

Allowed working time: 0.100 sec.

Allowed memory: 16.00 MB

Size limit: 16.00 KB

Checker: Case-Insensitive

[Python code](#)

[Submit](#)

Points	Time and memory used	Submission date
 100 / 100	Memory: 8.21 MB Time: 0.056 s	15:50:20 28.06.2018 <a href="#">Details</a>

## Задача: отлична оценка или не

Следващата задача от тази тема е да се напише **конзолна програма**, която **въвежда оценка** (десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока, или "Not excellent." в противен случай.

## Примерен вход и изход

Вход	Изход
6	Excellent!

Вход	Изход
5	Not Excellent!

Вход	Изход
5.49	Not excellent.

## Насоки и подсказки

Първо създаваме **нов Python файл** в нашия проект. Следва да **напишем кода** на програмата. Може да си помогнем със следния примерен код:

```
grade = float(input())
if grade >= 5.5:
    print("Excellent!")
else:
    print("Not excellent.")
```

Следва да **стартираме програмата**, както обикновено с [Shift + F10] и да я тестваме дали работи коректно:

```
Run: Debugging
4.25
Not excellent.

Process finished with exit code 0
```

```
Run: Debugging
5.6
Excellent!

Process finished with exit code 0
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#1>.

Submissions		
<b>1</b>	Points	Time and memory used
100 / 100	Memory: 8.27 MB Time: 0.056 s	Submission date 01:12:36 11.08.2018
<b>1</b>		

## Задача: четно или нечетно

Да се напише програма, която въвежда **цяло** число и печата дали е **четно** или **нечетно**.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	even	3	odd	25	odd	1024	even

### Насоки и подсказки

Отново, първо добавяме **нов Python файл**. Проверката дали дадено число е четно, може да се реализира с оператора **%**, който ще ни върне остатъка при целочислено деление на 2, по следния начин: **is\_even = number % 2 == 0**.

Остава да **стартираме** програмата с [Ctrl+F5] и да я тестваме:

```
Run: Debugging
42
even
Process finished with exit code 0
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#2>.

## Задача: намиране на по-голямото число

Да се напише програма, която въвежда **две цели числа** и отпечатва по-голямото от двете.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
5 3	5	3 5	5	10 10	10	-5 5	5

### Насоки и подсказки

Както обикновено, първо трябва да добавим **нов Python файл**. За кода на програмата ни е необходима единична **if-else** конструкция. Може да си помогнете частично с кода от картилката, който е умышлено замъглен, за да помисли читателя как да го допише сам:

```
print("Enter two integers: ")
firstNumber = int(input())
secondNumber = int(input())
```

След като сме готови с имплементацията на решението, **стартираме** програмата с [Shift + F10] и я тестваме:

```
Run: Debugging ×
▶ Enter two integers:
5
3
Bigger number: 5
Process finished with exit code 0
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#3>.

### Задача: изписване на число до 9 с думи

Да се напише програма, която въвежда **цяло** число в диапазона [0 ... 9] и го изписва с **думи** на английски език. Ако числото е извън диапазона, изписва "number too big".

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
5	five	1	one	9	nine	10	number too big

#### Насоки и подсказки

Може да използваме поредица **if-elif** конструкции, с които да разгледаме възможните **11** случая.

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#4>.

### Задача: познай паролата

Да се напише програма, която **въвежда парола** (произволен текст) и проверява дали въведеното **съвпада** с фразата "s3cr3t!P@ssw0rd". При съответствие да се изведе "Welcome", а при несъответствие да се изведе "Wrong password!".

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
qwerty	Wrong password!	s3cr3t!P@ssw0rd	Welcome	s3cr3t!p@ss	Wrong password!

### Насоки и подсказки

Може да използваме **if-else** конструкцията.

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#8>.

### Задача: число от 100 до 200

Да се напише програма, която **въвежда цяло число** и проверява дали е **под 100**, **между 100 и 200** или **над 200**. Да се отпечатат съответно съобщения, като в примерите по-долу.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
95	Less than 100	120	Between 100 and 200	210	Greater than 200

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1049#9>.

### Задача: еднакви думи

Да се напише програма, която **въвежда две думи** и проверява дали са еднакви. Да не се прави разлика между главни и малки букви. Да се изведе "yes" или "no".

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
Hello Hello	yes	SoftUni softuni	yes	Soft Uni	no	beer vodka	no

## Насоки и подсказки

Преди сравняване на думите, трябва да ги обърнем в долен регистър, за да не оказва влияние размера на буквите (главни/малки): **word = word.lower()**.

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#10>.

## Задача: информация за скоростта

Да се напише програма, която **въвежда скорост** (дробно число) и отпечатва **информация за скоростта**. При скорост до 10 (включително), отпечатайте "slow". При скорост над 10 и до 50, отпечатайте "average". При скорост над 50 и до 150, отпечатайте "fast". При скорост над 150 и до 1000, отпечатайте "ultra fast". При по-висока скорост, отпечатайте "extremely fast".

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
8	slow	126	fast	3500	extremely fast
49.5	average	160	ultra fast		

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#11>.

## Задача: лица на фигури

Да се напише програма, която **въвежда размерите на геометрична фигура** и **пресмята лицето ѝ**. Фигурите са четири вида: квадрат (**square**), правоъгълник (**rectangle**), кръг (**circle**) и триъгълник (**triangle**).

На първия ред на входа се чете вида на фигурата (**square**, **rectangle**, **circle**, **triangle**):

- Ако фигурата е **квадрат**, на следващия ред се чете едно число – дължина на страната му.
- Ако фигурата е **правоъгълник**, на следващите два реда се четат две числа – дълчините на страните му.
- Ако фигурата е **кръг**, на следващия ред се чете едно число – радиуса на кръга.
- Ако фигурата е **триъгълник**, на следващите два реда се четат две числа – дължината на страната му и дълчината на височината към нея.

Резултатът да се закръгли до 3 цифри след десетичния знак.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
square 5	25	rectangle 7 2.5	17.5	circle 6	113.097	triangle 4.5 20	45

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#12>.

## Задача: време + 15 минути

Да се напише програма, която **въвежда час и минути** от 24-часово денонощие и изчислява колко ще е **частът след 15 минути**. Резултатът да се отпечатва във формат **hh:mm**. Часовете винаги са между 0 и 23, а минутите винаги са между 0 и 59. Часовете се изписват с една или две цифри. Минутите се изписват винаги с по две цифри и с **водеща нула**, когато е необходимо.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1 46	2:01	0 01	0:16	23 59	0:14	11 08	11:23

## Насоки и подсказки

Добавете 15 минути и направете няколко проверки. Ако минутите надвишат 59, **увеличете часовете** с 1 и **намалете минутите** с 60. По аналогичен начин разгледайте случая, когато часовете надвишат 23. При печатането на минутите, проверете за водеща нула.

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#13>.

## Задача: еднакви 3 числа

Да се напише програма, в която се въвеждат **3 числа** и се отпечатва дали те са **еднакви** ("yes" / "no").

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
5 5 5	yes	5 4 5	no	1 2 3	no

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#14>.

### Задача: \* изписване на число от 0 до 100 с думи

Да се напише програма, която превръща число в диапазона [0 ... 100] в текст.

#### Примерен вход и изход

Вход	Изход
25	twenty five

Вход	Изход
42	forty two

Вход	Изход
6	six

#### Насоки и подсказки

Проверяваме първо за **едноцифрени числа** и ако числото е едноцифено, отпечатваме съответната дума за него. След това проверяваме за **двуцифрени числа**. Тях отпечатваме на две части: лява част (**десетици** = числото / 10) и дясна част (**единици** = числото % 10). Ако числото има 3 цифри, трябва да е 100 и може да се разгледа като специален случай.

## Тестване в Judge системата

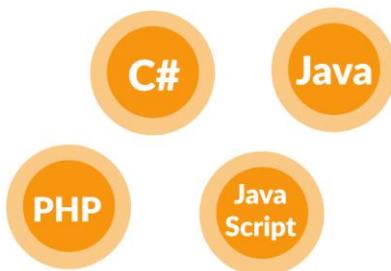
Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1049#15>.

Качествено образование,

професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 3.2. Прости проверки – изпитни задачи

В предходната глава разгледахме **условните конструкции** в езика Python, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Споменахме още какъв е обхватът на една променлива (нейният **scope**), както и как постъпково да проследяваме изпълнението на нашата програма (т.нар. **дебъгване**). В настоящата глава ще упражним работата с **логически проверки**, като разгледаме някои задачи, давани на изпити. За целта нека първо си припомним конструкцията на логическата проверка:

```
if булев израз:  
    # тяло на условната конструкция  
else:  
    # тяло на else-конструкция
```

**if** проверките се състоят от:

- **if** клауза + променлива от булев тип (**bool**) или булев логически израз (израз, който връща резултат **true/false**)
- тяло на конструкцията - съдържа произволен блок със сорс код
- **else** клауза и нейният блок със сорс код (**незадължително**)

## Изпитни задачи

След като си припомнихме как се пишат условни конструкции, да решим няколко задачи, за да получим практически опит с **if-else**-конструкцията.

### Задача: цена за транспорт

Студент трябва да пропътува **n** километра. Той има избор между **три вида транспорт**:

- **Такси.** Начална такса: 0.70 лв. Дневна тарифа: 0.79 лв./км. Нощна тарифа: 0.90 лв./км.
- **Автобус.** Дневна / нощна тарифа: 0.09 лв./км. Може да се използва за разстояния минимум 20 км.
- **Влак.** Дневна / нощна тарифа: 0.06 лв./км. Може да се използва за разстояния минимум 100 км.

Напишете програма, която въвежда броя **километри n** и **период от деня** (ден или нощ) и изчислява **цената на най-евтиния транспорт** измежду трите.

### Входни данни

От конзолата се четат **два реда**:

- Първият ред съдържа числото **n** – брой километри – цяло число в интервала [1 ... 5000].
- Вторият ред съдържа дума “**day**” или “**night**” – пътуване през деня или през нощта.

## Изходни данни

Да се отпечата на конзолата **най-ниската цена** за посочения брой километри.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
5 day	4.65	7 night	7	25 day	2.25	180 night	10.8

## Насоки и подсказки

Ще прочетем входните данни и в зависимост от разстоянието ще изберем най-евтиния транспорт. За целта ще използваме няколко проверки.

## Обработка на входните данни

В условието на задачата е дадена **информация за входа и изхода**. Съответно, първите **два реда** от решението ще съдържат декларирането и инициализирането на двете **променливи**, в които ще пазим **стойностите на входните данни**. За **първия ред** е **упоменато**, че съдържа **цяло число**, затова за първата променлива ще използваме стойност, преобразуване с функцията **int(...)**. За **втория ред** указанietо е, че съдържа **дума**, съответно променливата е от тип **string**:

```
distance = int(input())
day_or_night = input()
```

Преди да започнем проверките е нужно да **декларираме** и една **променлива**, в която ще пазим стойността на **цената за транспорт**:

```
price = 0.00
```

## Извършване на проверки и съответните изчисления

След като вече сме **декларирали** и **инициализирали** входните данни и променливата, в която ще пазим стойността на цената, трябва да преценим коя **условия** от задачата първо ще **бъдат проверени**.

От условието е видно, че тарифите на две от превозните средства **не зависят** от това дали е **ден** или **нощ**, но тарифата на единия превоз (такси) **зависи**. По тази причина **първата проверка** ще е именно дали е **ден** или **нощ**, за да стане ясно коя

тарифа на таксито ще се използва. За целта декларираме още една променлива, в която ще пазим стойността на тарифата на таксито:

```
taxi_rate = 0.00
```

За да изчислим тарифата на таксито, ще използваме проверка от типа **if-else**:

```
if day_or_night == 'day':
    taxi_rate = 0.79
else:
    taxi_rate = 0.90
```

След като е направено и това, вече може да пристъпим към изчислението на самата **цена за транспорта**. Ограниченията, които присъстват в условието на задачата, са относно **разстоянието**, което студента иска да пропътува. По тази причина, ще построим **if-elif-else** конструкция, с чиято помощ ще открием цената за транспорта в зависимост от подадените километри:

```
if distance < 20:
    price = 0.70 + distance * taxi_rate
elif distance < 100:
    price = distance * 0.09
else:
    price = distance * 0.06
```

Първо правим проверка дали километрите са **под 20**, тъй като от условието е видно, че **под 20** километра студента би могъл да използва само **такси**. Ако условието на проверката е **вярно** (връща **true**), на променливата, която пази стойността на цената на транспорта (**price**), ще присвоим съответната стойност. Тази стойност е равна на **първоначалната такса**, която **събираме** с неговата **тарифа**, **умножена по разстоянието**, което студента трябва да измине.

Ако условието на променливата **не е вярно** (връща **false**), следващата стъпка е програмата ни да провери дали километрите са **под 100**. Правим това, защото от условието е видно, че в този диапазон може да се използва и **автобус** като транспортно средство. **Цената** за километър на автобуса е **по-ниска** от тази на таксито. Следователно, ако резултата от проверката е **верен**, то в блок тялото на **elif**, на променливата за цената на транспорта (**price**) трябва да присвоим стойност, равна на резултата от **умножението** на **тарифата** на автобуса по **разстоянието**.

Ако и тази проверка **не даде true** като резултат, остава в тялото на **else** на променливата за цена да присвоим **стойност**, равна на **резултата** от **умножението** на **разстоянието по тарифата** на влака. Това се прави, тъй като влакът е **най-евтиния** вариант за транспорт при даденото разстояние.

## Отпечатване на изходните данни

След като сме направили **проверките** за разстоянието и сме **изчислили** цената на **най-евтиния транспорт**, следва да я **отпечатаме**. В условието на задачата няма

изисквания как да бъде форматиран резултата и по тази причина ще отпечатаме само **променливата**:

```
print(price)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1050#0>.

## Задача: тръби в басейн

Басейн с **обем V** има **две тръби**, от които се пълни. Всяка тръба има определен **дебит** (литрите вода, минаващи през една тръба за един час). Работникът пуска тръбите едновременно и излиза за **N часа**. Напишете програма, която изкарва състоянието на басейна, **в момента**, когато работникът се върне.

### Входни данни

От конзолата се четат **четири реда**:

- Първият ред съдържа числото **V** – обем на басейна в литри – цяло число в интервала [1 ... 10000].
- Вторият ред съдържа числото **P1** – дебит на първата тръба за час – цяло число в интервала [1 ... 5000].
- Третият ред съдържа числото **P2** – дебит на втората тръба за час – цяло число в интервала [1 ... 5000].
- Четвъртият ред съдържа числото **H** – часовете, в които работникът отсъства – число с плаваща запетая в интервала [1.0 ... 24.00].

### Изходни данни

Да се отпечата на конзолата **едно от двете възможни състояния**:

- До колко се е запълнил басейнът и коя тръба с колко процента е допринесла. Всички проценти да се форматират до цяло число (без закръгляне).
  - "The pool is [x]% full. Pipe 1: [y]%. Pipe 2: [z]%."
- Ако басейнът се е препълнил – с колко литра е прелял за даденото време, число с плаваща запетая.
  - "For [x] hours the pool overflows with [y] liters."

Имайте предвид, че поради закръглянето до цяло число се губят данни и е нормално сборът на процентите да е 99%, а не 100%.

### Примерен вход и изход

Вход	Изход	Вход	Изход
1000		100	
100	The pool is 66% full. Pipe 1: 45%. Pipe2: 54%.	100	
120		100	For 2.5 hours the pool overflows with 400 liters.
3		2.5	

## Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

### Обработка на входните данни

От условието на задачата виждаме, че в програмата ни трябва да има **четири реда**, от които четем **входните данни**. Чрез първите **три** ще въвеждаме **цели числа** и по тази причина стойностите на **променливите** ще бъдат от тип **int**. За **четвъртия** ред ни е казано, че ще бъде **число**, което е **с плаваща запетая**, затова и стойността на **променливата**, която ще използваме, ще е от тип **float**:

```
volume = [ ]
pipe_1 = [ ]
pipe_2 = [ ]
hours = [ ]
```

Следващата ни стъпка е да **декларираме и инициализираме** променлива, в която ще изчислим с колко **литра** се е **напълнил** басейна за **времето**, в което работникът е **отсъствал**. Изчисленията ще направим като **съберем** стойностите на дебита на **двете тръби** и ги **умножим по часовете**, които са ни зададени като вход:

```
water = [ ]
```

### Извършване на проверки и обработка на изходните данни

След като вече имаме и **стойността на количеството** вода, което е минало през **тръбите**, следва стъпката, в която трябва да **сравним** това количество с обема на самия басейн.

Ще ползваме пристрастна **if-else** проверка, в която условието ще е **дали количеството вода е по-малко от обема на басейна**. Ако проверката върне **true**, то трябва да разпечатаме един **ред**, който да съдържа в себе си **съотношението** между количеството **вода**, минало през **тръбите**, и **обема на басейна**, както и **съотношението** на количеството **вода** от всяка **една тръба** спрямо **обема на басейна**.

Съотношението е нужно да бъде изразено в **проценти**, затова и всички изчисления до момента в този ред ще бъдат **умножени по 100**. Стойностите ще бъдат вмъкнати с **placeholders** и тъй като има условие **результатата в проценти** да се форматира до **две цифри** след **десетичния** знак без **закръгление**, то за целта ще използваме функцията **math.trunc(...)**:

```

if water <= volume:
    print('The pool is {0}% full. Pipe 1: {1}%.'
          ' Pipe 2: {2}%.format(
    math.trunc(           ),
    math.trunc(           ),
    math.trunc(           )
))
else:
    print('For {0} hours the pool overflows '
          'with {1} liters.'.format(
    hours,
    water - volume
))

```

Ако проверката обаче върне резултат **false**, то това означава, че **количеството вода** е **по-голямо** от **обема** на басейна, съответно той е **прелял**. Отново изхода трябва да е на **един ред**, но този път съдържа в себе си само две стойности - тази на **часовете**, в които работникът е отсъствал, и **количеството вода**, което е разлика между влязлата вода и обема на басейна.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1050#1>.

## Задача: поспаливата котка Том

Котката Том обича по цял ден да спи, за негово съжаление стопанинът му си играе с него винаги когато има свободно време. За да се наспи добре, **нормата за игра** на Том е **30 000 минути в година**. Времето за игра на Том **зависи от почивните дни на стопанина му**:

- Когато е на **работа**, стопанинът му си играе с него **по 63 минути на ден**.
- Когато **почива**, стопанинът му си играе с него **по 127 минути на ден**.

Напишете програма, която въвежда **броя почивни дни** и отпечатва дали Том може да се наспи добре и колко е **разликата от нормата** за текущата година, като приемем че **годината има 365 дни**.

**Пример:** 20 почивни дни -> работните дни са 345 ( $365 - 20 = 345$ ). Реалното време за игра е 24 275 минути ( $345 * 63 + 20 * 127$ ). Разликата от нормата е 5 725 минути ( $30\ 000 - 24\ 275 = 5\ 725$ ) или 95 часа и 25 минути.

## Входни данни

Входът се чете от конзолата и се състои от едно цяло число - **броят почивни дни** в интервала [0 ... 365].

## Изходни данни

На конзолата трябва да се отпечатат **два реда**.

- Ако времето за игра на Том е **над нормата** за текущата година:
  - На **първия ред** отпечатайте: "Tom will run away".
  - На **втория ред** отпечатайте разликата от нормата във формат: "{H} hours and {M} minutes more for play".
- Ако времето за игра на Том е **под нормата** за текущата година:
  - На **първия ред** отпечатайте: "Tom sleeps well".
  - На **втория ред** отпечатайте разликата от нормата във формат: "{H} hours and {M} minutes less for play".

## Примерен вход и изход

Вход	Изход	Вход	Изход
20	Tom sleeps well 95 hours and 25 minutes less for play	113	Tom will run away 3 hours and 47 minutes for play

## Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

### Обработка на входните данни и прилежащи изчисления

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от **един ред**, който ще съдържа в себе си **едно цяло число** в интервала [0 ... 365]. По тази причина на променливата ще присвоим стойност от тип **int**:

`holidays =`

За да решим задачата, **първо** трябва да изчислим колко **общо минути** стопанинът на Том си играе с него. От условието виждаме, че освен в **почивните дни**, спасявата котка трябва да си играе и в **работните** за стопанина му. Числото, което прочитаме от конзолата, е това на **почивните дни**.

Следващата ни стъпка е с помощта на това число да **изчислим** колко са **работните дни** на стопанина, тъй като без тях не можем да стигнем до **общото количество минути за игра**. Щом общият брой на дните в годината е **365**, а броят на почивните дни е **X**, то това означава, че броят на работните дни е **365 - X**. **Разликата** ще запазим в нова променлива, която ще използваме **само** за тази стойност:

`working_days =`

След като вече имаме **количествата дни за игра**, то вече можем да **изчислим времето за игра** на Том в минути. Неговата **стойност е равна** на резултата от умножението на **работните дни по 63** минути (в условието е зададено, че в работни дни, времето за игра е 63 минути на ден) **събран с резултата от умножението на почивните дни по 127** минути (в условието е зададено, че в почивните дни, времето за игра е 127 минути на ден):

```
total_play_minutes =
```

В условието на задачата за изхода виждаме, че ще трябва да **разпечатаме разликата** между двете стойности в **часове и минути**. За тази цел от **общото** време за игра ще **извадим** нормата от **30 000** минути и получената разлика ще **запищем** в **нова** променлива. След това тази променлива ще **разделим целочислено** на 60, за да получим **часовете**, а след това, за да открием колко са **минутите** ще използваме **модулно деление с оператора %**, като отново ще разделим променливата на разликата с 60.

Тук трябва да отбележим, че ако полученото количество **време за игра** на Том е **по-малко** от **30 000**, при **изваждането** на нормата от него ще получим **число с отрицателен знак**. За да **неутрализираме** знака в двете деления по-късно, ще използваме **функцията math.fabs(...)** при намирането на разликата:

```
difference = math.fabs( )
hours =
minutes =
```

## Извършване на проверки

Времето за игра вече е изчислено, което ни води до **следващата** стъпка - **сравняване на времето за игра** на Том с **нормата**, от която зависи дали котката може да се наспива добре. За целта ще използваме **if-else** проверка, като в **if** клаузата ще проверим дали времето за игра е **по-голямо** от **30 000** (нормата).

## Обработка на изходните данни

Какъвто и резултат да ни върне проверката, то трябва да разпечатаме колко е **разликата в часове и минути**. Това ще направим с **placeholder** и променливите, в които изчислихме стойностите на часовете и минутите, като форматирането ще е според условието за изход:

```
if total_play_minutes > 30000:
    print('Tom will run away')
    print(f'{hours} hours and {minutes} minutes more for play')

else:
    print('Tom sleeps well')
    print(f'{hours} hours and {minutes} minutes less for play')
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1050#2>.

### Задача: реколта

От лозе с площ  $X$  квадратни метри се заделя 40% от реколтата за производство на вино. От 1 кв.м. лозе се изкарват  $Y$  килограма грозде. За 1 литър вино са нужни 2,5 кг. грозде. Желаното количество вино за продан е  $Z$  литра.

Напишете програма, която пресмята колко вино може да се произведе и дали това количество е достатъчно. Ако е достатъчно, остатъкът се разделя по равно между работниците на лозето.

#### Входни данни

Входът се чете от конзолата и се състои от **точно 4 реда**:

- 1-ви ред:  $X$  кв.м е лозето – цяло число в интервала [10 ... 5000].
- 2-ри ред:  $Y$  грозде за един кв.м. – реално число в интервала [0.00 ... 10.00].
- 3-ти ред:  $Z$  нужни литри вино – цяло число в интервала [10 ... 600].
- 4-ти ред: брой работници – цяло число в интервала [1 ... 20].

#### Изходни данни

На конзолата трябва да се отпечата следното:

- Ако произведеното вино е **по-малко от нужното**:
  - "It will be a tough winter! More {недостигащо вино} liters wine needed."  
\* Резултатът трябва да е закръглен към по-ниско цяло число.
- Ако произведеното вино е **повече от нужното**:
  - "Good harvest this year! Total wine: {общо вино} liters."  
\* Резултатът трябва да е закръглен към по-ниско цяло число.
  - "{Оставащо вино} liters left -> {вино за 1 работник} liters per person."  
\* И двата резултата трябва да са закръглени към по-високото цяло число.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
650	Good harvest this year! Total wine: 208 liters.	1020	
2		1.5	It will be a tough winter! More
175	33 liters left -> 11 liters per person.	425	180 liters wine needed.
3		4	

## Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

### Обработка на входните данни и прилежащи изчисления

Първо трябва да проверим какви ще са **входните данни** и да изберем какви **променливи** ще използваме. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
vineyard_area = ...
grape_per_square = ...
needed_liters =
workers = ...
```

За да решим задачата е нужно да **изчислим** колко **литра вино** ще получим на база **входните данни**. От условието на задачата виждаме, че за да **пресметнем** количеството **вино в литри**, трябва първо да разберем какво е **количеството грозде в килограми**, което ще се получи от тази реколта. За тази цел ще **декларираме** една **променлива**, на която ще присвоим **стойност**, равна на **40%** от резултата от **умножението** на площта на лозето и количеството грозде, което се получава от 1 кв. м.

След като сме извършили тези пресмятания, сме готови да **пресметнем** и **количеството вино в литри**, което ще се получи от тази реколта. За тази цел **декларираме** още една **променлива**, в която ще пазим това **количество**, а от условието стигаме до извода, че за да го пресметнем, е нужно да **разделим** количеството грозде в **кг** на **2.5**:

```
harvest_per_vine =
vine = ...
```

### Извършване на проверки и обработка на изходните данни

Вече сме направили нужните пресмятания и следващата стъпка е да **роверим** дали получените литри вино са **достатъчни**. За целта ще използваме **проста условна конструкция** от типа **if-else**, като в условието ще проверим дали литрите вино от реколтата са **повече от** или **равни** на **нужните литри**.

Ако проверката върне резултат **true**, от условието на задачата виждаме, че на **първия ред** трябва да разпечатаме **виното**, което сме **получили** от реколтата. За да спазим условието, **тази стойност** да бъде **закръглена** до по-ниското **цяло число**, ще използваме функцията **math.floor(...)** при разпечатването й чрез **placeholder**.

На **втория ред** има изискване да разпечатаме резултатите, като ги **закръглим** към **по-високото цяло число**, което ще направим с функцията **math.ceil(...)**. Стойностите, които трябва да разпечатаме, са на **оставащото количество вино** и **количеството вино**, което се пада на **един работник**. Оставащото количество вино

е равно на **разликата** между получените литри вино и нужните литри вино. Стойността на това количество ще изчислим в нова променлива, която ще декларираме и инициализираме в **блок тялото** на **if**, преди разпечатването на първия ред. Количество вино, което се полага на **един работник**, ще изчислим като оставащото вино го разделим на броя на работниците:

```
if :  
    vine_left =  
    print('Good harvest this year! Total wine: {0} liters.' . format(  
        math.floor(      )))  
    print('{0} liters left -> {1} liters per person.' . format(  
        math.ceil(      ), math.ceil(      )))
```

Ако проверката ни върне резултат **false** от условието на задачата виждаме, че трябва да разпечатаме разликата от нужните литри и получените от тази реколта **литри вино**. Има условие резултата да е закръглен към по-ниското цяло число, което ще направим с функцията **math.floor(...)**:

```
else:  
    print('It will be a tough winter! More '  
          '{0} liters wine needed.' . format(  
              math.floor((      ))))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1050#3>.

## Задача: фирма

Фирма получава заявка за изработването на проект, за който са необходими определен брой часове. Фирмата разполага с **определен брой дни**. През 10% от дните служителите са на **обучение** и не могат да работят по проекта. Един нормален **работен ден** във фирмата е **8 часа**. Проектът е важен за фирмата и всеки служител задължително работи по проекта в **извънработно време** по **2 часа / ден**.

Часовете трябва да са **закръглени към по-ниско цяло число** (например → **6.98 часа** се закръглат на **6 часа**).

Напишете програма, която изчислява дали фирмата може да завърши проекта навреме и колко часа не достигат или остават.

## Входни данни

Входът се чете от конзолата и съдържа точно 3 реда:

- На първия ред са **необходимите часове** – цяло число в интервала [0 ... 200 000].

- На втория ред са дните, с които фирмата разполага – цяло число в интервала [0 ... 20 000].
- На третия ред е броят на всички служители – цяло число в интервала [0 ... 200].

## Изходни данни

Да се отпечата на конзолата един ред:

- Ако времето е достатъчно: "Yes!{оставащите часове} hours left."
- Ако времето НЕ Е достатъчно: "Not enough time!{недостигащите часове} hours needed."

## Примерен вход и изход

Вход	Изход	Вход	Изход
90 7 3	Yes!99 hours left.	99 3 1	Not enough time!72 hours needed.

## Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

## Обработка на входните данни

За решението на задачата е нужно **първо** да прочетем **входните данни**. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
project_hours = ...
available_days = ...
overtime_workers = ...
```

## Помощни изчисления

Следващата стъпка е да изчислим **количество** на работните часове като умножим работните дни по 8 (всеки ден се работи по 8 часа) с броя на работниците и ги съберем с извънработното време. **Работните дни** са равни на **90% от дните**, с които фирмата разполага. **Извънработното време** е равно на резултата от умножението на броя на служителите с 2 (възможните часове извънработно време), като това също се умножава по броя на дните, с които фирмата разполага. От условието на задачата виждаме, че има условие **часовете да са закръглени към по-ниско цяло число**, което ще направим с функцията **math.floor(...)**:

```
work_days = int(input("Enter work days:"))
overtime_hours = int(input("Enter overtime hours:"))
work_hours = int(input("Enter work hours:"))
total_hours = int(input("Enter total hours:"))
```

## Извършване на проверки

След като сме направили изчисленията, които са ни нужни за да разберем стойността на **работните часове**, следва да направим проверка дали тези часове **достигат или остават допълнителни** такива.

Ако **времето е достатъчно**, разпечатваме резултата, който се изисква в условието на задачата, а именно разликата между **работните часове и необходимите часове** за завършване на проекта.

Ако **времето не е достатъчно**, разпечатваме допълнителните часове, които са нужни за завършване на проекта и са равни на разликата между **часовете за проекта и работните часове**:

```
if total_hours - work_hours - overtime_hours >= 0:
    print('Yes! {0} hours left.'.format(
        total_hours - work_hours - overtime_hours))
else:
    print('Not enough time! {0} hours needed.'.format(
        work_hours + overtime_hours - total_hours))
```

## Тестване в Judge системата

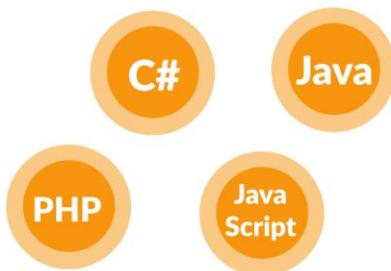
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1050#4>.

Качествено образование,

професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 4.1. По-сложни проверки

В настоящата глава ще разгледаме вложените проверки в езика Python, чрез които нашата програма може да съдържа **условни конструкции**, в които има **вложени други условни конструкции**. Наричаме ги "вложени", защото поставяме **if** конструкция в друга **if** конструкция. Ще разгледаме и **по-сложни логически условия** с подходящи примери.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=aeeje4nlzas>.

## Вложени проверки

Доста често програмната логика налага използването на **if** или **if-else** конструкции, които се съдържат една в друга. Те биват наричани **вложени if** или **if-else** конструкции. Както се подразбира от названието "вложени", това са **if** или **if-else** конструкции, които са поставени в други **if** или **else** конструкции.

```
if condition1:  
    if condition2:  
        # тяло  
    else:  
        # тяло
```

Влагането на повече от три условни конструкции една в друга не се счита за добра практика и трябва да се избягва, най-вече чрез оптимизиране на структурата / алгоритъма на кода и/или чрез използването на друг вид условна конструкция, който ще разгледаме по-надолу в тази глава.

## Пример: обръщение според възраст и пол

Според въведени **възраст** (десетично число) и **пол** (**m** / **f**) да се отпечата обръщение:

- "Mr." – мъж (пол "m") на 16 или повече години.
- "Master" – момче (пол "m") под 16 години.
- "Ms." – жена (пол "f") на 16 или повече години.
- "Miss" – момиче (пол "f") под 16 години.

## Примерен вход и изход

Вход	Изход
12 f	Miss

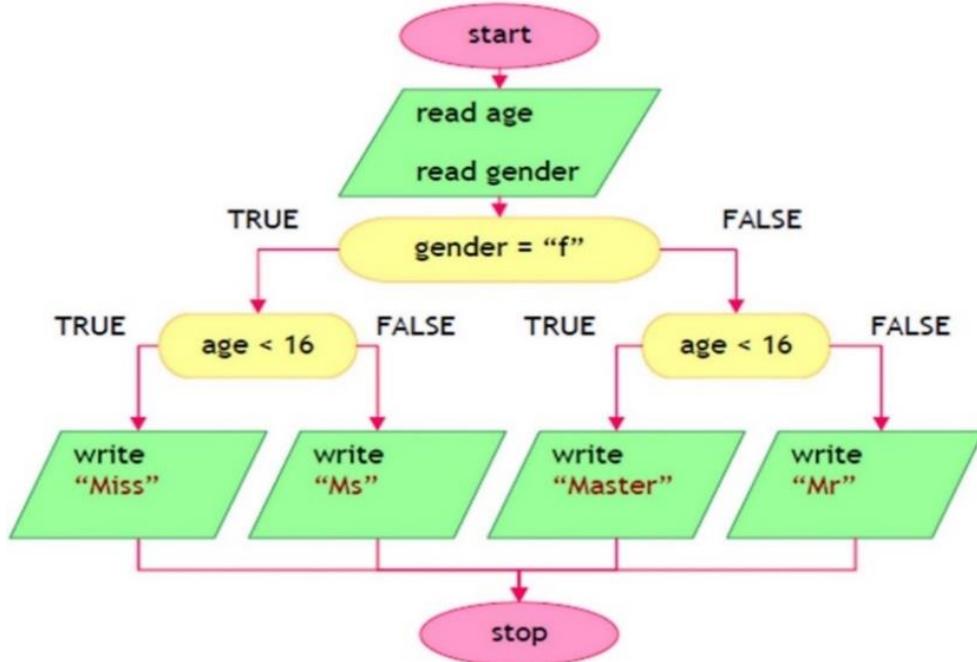
Вход	Изход
17 m	Mr.

Вход	Изход
25 f	Ms.

Вход	Изход
13.5 m	Master

## Решение

Можем да забележим, че **изходът** на програмата зависи от няколко неща. Първо трябва да проверим какъв **пол** е въведен и **после** да проверим **възрастта**. Съответно ще използваме няколко **if-else** блока. Тези блокове ще бъдат вложени, т.е. от **результатата** на първия ще се определи кои от **другите** да се изпълни:



След прочитане на входните данни от конзолата ще трябва да се изпълни следната примерна програмна логика:

```

age = float(input())
gender = str(input())

if age < 16:
    if gender == 'm':
        print('Master')
    elif gender == 'f':
        print('Miss')
else:
    if gender == 'm':
        print('Mr.')
    elif gender == 'f':
        print('Ms.')
  
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#0>.

## Пример: квартално магазинче

Предприемчив българин отваря по едно **квартално магазинче** в няколко града с различни **цени** за следните **продукти**:

продукт / град	Sofia	Plovdiv	Varna
coffee	0.50	0.40	0.45
water	0.80	0.70	0.70
beer	1.20	1.15	1.10
sweets	1.45	1.30	1.35
peanuts	1.60	1.50	1.55

По даден **град** (стринг), **продукт** (стринг) и **количество** (десетично число) да се пресметне цената.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
coffee		peanuts		beer		water	
Varna	0.9	Plovdiv	1.5	Sofia	7.2	Plovdiv	2.1
2		1		6		3	

### Решение

Прехвърляме всички букви в **долен регистър** с функцията **.lower()**, за да сравняваме продукти и градове **без значение** от малки/главни букви:

```
product = input().lower()
town = input().lower()
quantity = float(input())

if town == 'sofia':
    if product == 'coffee':
        print(0.50 * quantity)
    # TODO: Finish this ...

if town == 'varna':
    # TODO Finish this ...

if town == 'plovdiv':
    # TODO Finish this ...
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#1>.

## По-сложни проверки

Нека разгледаме как можем да правим по-сложни логически проверки. Може да използваме логическо "И" (**and**), логическо "ИЛИ" (**or**), логическо **отрицание** (**not**) и скоби () .

### Логическо "И"

Както видяхме, в някои задачи се налага да правим **много проверки наведнъж**. Но какво става, когато за да изпълним някакъв код, трябва да бъдат изпълнени **няколко условия едновременно** и **неискаме да правим отрицание** (**else**) за всяко едно от тях? Вариантът с вложените **if блокове** е валиден, но кодът би изглеждал много **неподреден** и със сигурност **труден** за четене и поддръжка.

Логическо "И" (оператор **and**) означава няколко условия да са **изпълнени едновременно**. В сила е следната таблица на истинност:

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

### Как работи операторът **and**?

Операторът **and** приема **няколко булеви** (условни) израза, които имат стойност **True** или **False**, и ни връща **един** булев израз като **результат**. Използването му вместо редица вложени **if** блокове прави кода **по-четлив**, **подреден** и **лесен** за поддръжка. Но как работи, когато поставим **няколко** условия едно след друго? Както видяхме по-горе, логическото "И" връща **True**, **само** когато приема като **аргументи изрази** със стойност **True**. Съответно, когато имаме **последователност** от аргументи, логическото "И" **проверява** или докато **свършат** аргументите, или докато не срещне аргумент със стойност **False**.

Пример:

```
a = True
b = True
c = False
d = True

result = a and b and c and d
# False (като d не се проверява)
```

Програмата ще се изпълни по **следния** начин: започва проверката от **a**, прочита я и отчита, че има стойност **True**, след което **проверява b**. След като е **отчела**, че **a** и **b** връщат стойност **True**, **проверява следващия** аргумент. Стига до **c** и отчита, че променливата има стойност **False**. След като програмата отчете, че аргументът **c** има стойност **False**, тя изчислява израза **до c, независимо** каква е стойността на **d**. За това проверката на **d** се **прескача** и целият израз бива изчислен като **False**.

## Пример: точка в правоъгълник

Проверка дали точка  $\{x, y\}$  се намира **вътре** в правоъгълника  $\{x_1, y_1\} - \{x_2, y_2\}$ . Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа  $x_1, y_1, x_2, y_2, x$  и  $y$  (като се гарантира, че  $x_1 < x_2$  и  $y_1 < y_2$ ).

### Примерен вход и изход

Вход	Изход	Визуализация
2 -3 12 3 8 -1	Inside	<p>The diagram shows a Cartesian coordinate system with x-axis ticks at 0, 2, 4, 6, 8, 10, and 12. The y-axis ticks are -5, -3, -1, 1, and 3. A blue rectangle is drawn with vertices at (2, -3), (8, -3), (8, 3), and (2, 3). Inside the rectangle, there is a point labeled <math>x, y</math> located roughly at (5, 0.5). The rectangle is shaded in light blue.</p>

### Решение

Една точка е вътрешна за даден многоъгълник, ако **едновременно** са изпълнени следните четири условия:

- Точката е надясно от лявата страна на правоъгълника.
- Точката е наляво от дясната страна на правоъгълника.
- Точката е надолу от горната страна на правоъгълника.
- Точката е нагоре от долната страна на правоъгълника.

```

x1 = float(input())
y1 = float(input())
x2 = float(input())
y2 = float(input())

x = float(input())
y = float(input())

```

```

if x >= x1 and x <= x2 and y >= y1 and y <= y2:
    print('Inside')
else:
    print('Outside')

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#2>.

## Логическо "ИЛИ"

Логическо "ИЛИ" (оператор **or**) означава да е изпълнено поне едно измежду няколко условия. Подобно на оператора **and**, логическото "ИЛИ" приема няколко аргумента от булев (условен) тип и връща **True** или **False**. Лесно можем да се досетим, че получаваме като стойност **True**, винаги когато поне един от аргументите има стойност **True**. Типичен пример за логиката на този оператор е следният:

В училище учителят казва: "Иван или Петър да измият дъската". За да бъде изпълнено това условие (дъската да бъде измита), е възможно само Иван да я измие, само Петър да я измие или и двамата да го направят.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

## Как работи операторът **or** ?

Вече научихме какво представя логическото "ИЛИ". Но как всъщност се реализира? Както при логическото "И", програмата **проверява** от ляво на дясно **аргументите**, които са зададени. За да получим **True** от израза, е необходимо **само един** аргумент да има стойност **True**, съответно проверката **продължава** докато се срещне **аргумент с такава** стойност или докато **не свършат** аргументите.

Ето един **пример** за оператора **or** в действие:

```

a = False
b = True
c = False
d = True

result = a or b or c or d
# True (като с и d не се проверяват)

```

Програмата **проверява** **a**, отчита, че има стойност **False** и продължава. Стигайки до **b**, отчита, че има стойност **True** и целият израз получава стойност **True**, **без** да се проверява **c** и **d**, защото техните стойности **не биха променили** резултата на израза.

## Пример: плод или зеленчук

Нека проверим дали даден продукт е **плод** или **зеленчук**. Плодовете "fruit" са banana, apple, kiwi, cherry, lemon и grapes. Зеленчуците "vegetable" са tomato, cucumber, pepper и carrot. Всички останали са "unknown".

### Примерен вход и изход

Вход	Изход
banana	fruit
tomato	vegetable
java	unknown

### Решение

Трябва да използваме няколко условни проверки с логическо "ИЛИ" (**or**):

```
s = input()

if (s == 'banana' or s == 'apple'
    or s == 'kiwi' or s == 'cherry'
    or s == 'lemon' or s == 'grapes'):
    print('fruit')
elif s == 'tomato' or s == 'cucumber' \
    or s == 'pepper' or s == 'carrot':
    print('vegetable')
else:
    print('unknown')
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#3>.

## Логическо отрицание

Логическо отрицание (оператор **not**) означава да **не е изпълнено** дадено условие.

a	not a
True	False

Операторът **not** приема като **аргумент** булева променлива и **обръща** стойността ѝ (истината става лъжа, а лъжата става истина).

## Пример: невалидно число

Дадено **число е валидно**, ако е в диапазона [100 ... 200] или е 0. Да се направи проверка за **невалидно** число.

### Примерен вход и изход

Вход	Изход
75	invalid
150	(няма изход)
220	invalid

### Решение

```
num = float(input())
in_range = (num >= 100 and num <= 200) or num == 0
if not in_range:
    print('invalid')
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#4>.

## Операторът скоби ()

Както останалите оператори в програмирането, така и операторите **and** и **or** имат приоритет, като в случая **and** е с по-голям приоритет от **or**. Операторът **()** служи за **промяна на приоритета на операторите** и се изчислява пръв, също както в математиката. Използването на скоби също така придава по-добра четимост на кода и се счита за добра практика.

## По-сложни логически условия

Понякога условията може да са доста сложни, така че да изискват дълъг булев израз или поредица от проверки. Да разгледаме няколко такива примера.

## Пример: точка върху страна на правоъгълник

Да се напише програма, която проверява дали точка {x, y} се намира **върху** **някоя от страните** на правоъгълник {x1, y1} - {x2, y2}. Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа x1, y1, x2, y2, x и y (като се гарантира, че  $x_1 < x_2$  и  $y_1 < y_2$ ). Да се отпечата "Border" (точката лежи на някоя от страните) или "Inside / Outside" (в противен случай).

## Примерен вход и изход

Вход	Изход	Вход	Изход	
2		2		0 2 4 6 8 10 12
-3		-3		-5 -3 -1 1 3 5
12	Border	12	Inside / Outside	x1, y1
3		3		x, y
12		8		x2, y2
-1		-1		

## Решение

Точка лежи върху някоя от страните на правоъгълник, ако:

- $x$  съвпада с  $x_1$  или  $x_2$  и същевременно  $y$  е между  $y_1$  и  $y_2$  или
- $y$  съвпада с  $y_1$  или  $y_2$  и същевременно  $x$  е между  $x_1$  и  $x_2$ .

```
if ((x == x1 or x == x2) and (y >= y1) and (y <= y2))  
    or ((y == y1 or y == y2) and (x >= x1) and (x <= x2)):  
    print ('Border')
```

Предходната проверка може да се опрости по този начин:

```
on_left_side = (x == x1) and (y >= y1) and (y <= y2)  
on_right_side = (x == x2) and (y >= y1) and (y <= y2)  
on_up_side = (y == y1) and (x >= x1) and (x <= x2)  
on_down_side = (y == y2) and (x >= x1) and (x <= x2)  
  
if on_left_side or on_right_side or on_up_side or on_down_side:  
    print('Border')  
else:  
    print('Inside / Outside')
```

Вторият начин с допълнителните булеви променливи е по-дълъг, но е много по-разбираем от първия, нали? Препоръчваме ви когато пишете булеви условия, да ги правите **лесни за четене и разбиране**, а не кратки. Ако се налага, ползвайте допълнителни променливи със смислени имена. Имената на булевите променливи трябва да подсказват каква стойност се съхранява в тях.

Остава да се допише кода, за да отпечатва "Inside / Outside", ако точката не е върху някоя от страните на правоъгълника.

## Тестване в Judge системата

След като допишете решението, може да го тествате тук:  
<https://judge.softuni.bg/Contests/Practice/Index/1051#5>.

## Пример: магазин за плодове

Магазин за плодове в **работни дни** продава на следните **цени**:

Плод	Цена
banana	2.50
apple	1.20
orange	0.85
grapefruit	1.45
kiwi	2.70
pineapple	5.50
grapes	3.85

В почивни дни цените са **по-високи**:

Плод	Цена
banana	2.70
apple	1.25
orange	0.90
grapefruit	1.60
kiwi	3.00
pineapple	5.60
grapes	4.20

Напишете програма, която чете от конзолата **плод** (banana / apple / ...), **ден от седмицата** (Monday / Tuesday / ...) и **количество** (десетично число) и пресмята цената според цените от таблиците по-горе. Резултатът да се отпечата **закръглен** с 2 цифри след десетичния знак. При **невалиден ден** от седмицата или **невалидно име**на плод да се отпечата "error".

### Примерен вход и изход

Вход	Изход
orange	
Sunday	2.70

Вход	Изход
kiwi	
Monday	6.75

Вход	Изход
grapes	
Saturday	2.10

Вход	Изход
tomato	
Monday	error

### Решение

```
if (day == "saturday" or day == "sunday"):
    if (fruit == "banana"):
        price = 2.70
    elif (fruit == "apple"):
        price = 1.25
    # TODO: more fruits come here ...
elif (day == "monday" or day == "tuesday"
     or day == "wednesday" or day == "thursday"
     or day == "friday"):
    if (fruit == "banana"):
        price = 2.50
    # TODO: more fruits come here ...
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#6>.

## Пример: търговски комисионни

Фирма дава следните **комисионни** на търговците си според **града**, в който работят и обема на продажбите s:

Град	$0 \leq s \leq 500$	$500 < s \leq 1000$	$1000 < s \leq 10000$	$s > 10000$
Sofia	5%	7%	8%	12%
Varna	4.5%	7.5%	10%	13%
Plovdiv	5.5%	8%	12%	14.5%

Напишете програма, която чете име на **град** (стринг) и обем на **продажбите** (десетично число) и изчислява размера на комисионната. Резултатът да се изведе закръглен с **2 десетични цифри след десетичния знак**. При **невалиден град или обем на продажбите** (отрицателно число) да се отпечата "error".

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
Sofia 1500	120.00	Plovdiv 499.99	27.50	Kaspichan -50	error

### Решение

При прочитането на входа можем да обърнем града в малки букви (с функцията **.lower()**). Първоначално задаваме комисионната да е **-1**. Тя ще бъде променена, ако градът и ценовият диапазон бъдат намерени в таблицата с комисионните. За да изчислим комисионната според града и обема на продажбите, се нуждаем от няколко вложени **if проверки**, както е в примерния код по-долу:

```

if town == "sofia":
    if 0 <= sales and sales <= 500:
        comission = 0.05
    elif 500 < sales and sales <= 1000:
        comission = 0.07
    # TODO: check the other price ranges ...
elif town == "varna":
    # TODO: check the price ranges ...
elif town == "plovdiv":
    # TODO: check the price ranges ...

if comission >= 0:
    print("% .2f" % (sales * comission))
else:
    print("error")

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#7>.

## Пример: ден от седмицата

Нека напишем програма, която принтира **дения от седмицата** (на английски) според въведеното число (1 ... 7) или "Error", ако е подаден невалиден ден.

### Примерен вход и изход

Вход	Изход
1	Monday
7	Sunday
-1	Error

### Решение

```
day = int(input())

if day == 1:
    print("Monday")
elif day == 2:
    print("Tuesday")
...
elif day == 7:
    print("Sunday")
else:
    print("Error")
```



Добра практика е на първо място да поставяме онези условия, които обработват най-често случилите се ситуации, а конструкциите, обработващи по-рядко възникващи ситуации, да оставим в края на конструкцията.

Друга добра практика е да подреждаме случаите в нарастващ ред, без значение дали са целочислени или символни.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#8>.

## Пример: вид животно

Напишете програма, която принтира вида на животно според името му:

- dog → mammal

- crocodile, tortoise, snake → **reptile**
- others → **unknown**

### Примерен вход и изход

Вход	Изход
tortoise	reptile

Вход	Изход
dog	mammal

Вход	Изход
elephant	unknown

### Решение

Можем да решим задачата чрез няколко **if-elif** проверки по следния начин:

```
if animal == "dog":
    print("mammal")
elif animal == "crocodile" or animal == "tortoise" \
    or animal == "snake":
    print("reptile")
else:
    print("unknown")
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1051#9>.

## Какво научихме от тази глава?

Да си припомним новите конструкции и програмни техники, с които се запознахме в тази глава:

### Вложени проверки

```
if condition1:
    if condition2:
        # тяло
    else:
        # тяло
```

## По-сложни проверки с and, or, not и ()

```
if (x == left or x == right) and (y >= top or y <= bottom):
    print(...)
```

## Упражнения: по-сложни проверки

Нека сега да упражним работата с по-сложни проверки. Да решим няколко практически задачи.

## Задача: кино

В една кинозала столовете са наредени в **правоъгълна** форма в **r** реда и **c** колони. Има три вида прожекции с билети на **различни** цени:

- **Premiere** – премиерна прожекция, на цена **12.00** лева.
- **Normal** – стандартна прожекция, на цена **7.50** лева.
- **Discount** – прожекция за деца, ученици и студенти на намалена цена от **5.00** лева.

Напишете програма, която въвежда **тип прожекция** (стринг), брой **редове** и брой **колони** в залата (цели числа) и изчислява **общите приходи** от билети при **пълна зала**. Резултатът да се отпечата във формат като в примерите по-долу - с 2 цифри след десетичния знак.

### Примерен вход и изход

Вход	Изход	Вход	Изход
Premiere		Normal	
10	1440.00 leva	21	
12		13	2047.50 leva

### Насоки и подсказки

При прочитането на входа можем да обърнем типа на прожекцията в малки букви (с функцията **.lower()**). Създаваме и променлива, която ще ни съхранява изчислените приходи. В друга променлива пресмятаме пълния капацитет на залата. Използваме **if-elif** условна конструкция, за да изчислим прихода в зависимост от вида на прожекцията и отпечатваме резултата на конзолата в зададения формат (потърсете нужната **Python** функционалност в интернет).

Примерен код на описаната идея (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
type = input().lower()
rows =
columns =

full =
income =

if type == "premiere":
    income = full * 12.00
```

```
def main():
    print("Hello world")
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1051#10>.

### Задача: волейбол

Влади е студент, живее в София и си ходи от време на време до родния град. Той е много запален по волейбала, но е зает през работните дни и играе **волейбол** само през **уикендите** и в **празничните дни**. Влади играе в **София** всяка събота, когато **не е на работа** и не си пътува до **родния град**, както и в **2/3 от празничните дни**. Той пътува до **родния си град h** пъти в годината, където играе волейбол със старите си приятели в **неделя**. Влади **не е на работа 3/4 от уикендите**, в които е в София. Отделно, през **високосните години** Влади играе с **15%** повече волейбол от нормалното. Приемаме, че годината има точно **48 уикенда**, подходящи за волейбол. Напишете програма, която изчислява **колко пъти Влади е играл волейбол** през годината. **Закръглете резултата** надолу до най-близкото цяло число (напр. 2.15 -> 2; 9.95 -> 9).

Входните данни се четат от конзолата:

- Първият ред съдържа думата “**leap**” (високосна година) или “**normal**” (нормална година с 365 дни).
- Вторият ред съдържа цялото число **p** – брой празници в годината (които не са събота или неделя).
- Третият ред съдържа цялото число **h** – брой уикенди, в които Влади си пътува до родния град.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
leap		normal		normal		leap	
5	45	3	38	11	44	0	41
2		2		6		1	

### Насоки и подсказки

Стандартно прочитаме входните данни от конзолата като за избягване на грешки при въвеждане обръщаме текста в малки букви с функцията **.lower()**. Последователно пресмятаме **уикендите прекарани в София**, времето за игра в София и

**общото време за игра.** Накрая проверяваме дали годината е **високосна**, правим допълнителни изчисления при необходимост и извеждаме резултата на конзолата **закръглен надолу** до най-близкото **цяло число** (потърсете **Python** функция с такава функционалност в интернет).

Примерен код на описаната идея (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
year = input().lower()
holidays =
weekends_home =
sofia_weekends =
play_sofia =
play_total =

if year == "leap":
    play_total =
elif year == "normal":
    play_total =

print(play_total)
```

## Тестване в Judge системата

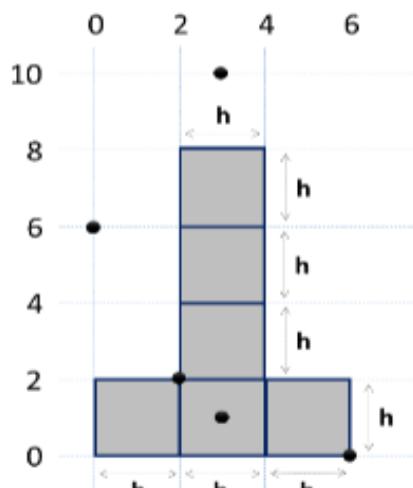
Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1051#11>.

### Задача: \* точка във фигурата

Фигура се състои от 6 блокчета с размер  $h \times h$ , разположени като на фигурата. Долният ляв ъгъл на сградата е на позиция  $\{0, 0\}$ . Горният десен ъгъл на фигурата е на позиция  $\{2*h, 4*h\}$ . На фигурата координатите са дадени при  $h = 2$ .

Да се напише програма, която въвежда цяло число  $h$  и координатите на дадена **точка**  $\{x, y\}$  (цели числа) и отпечатва дали точката е вътре във фигурата (**inside**), вън от фигурата (**outside**) или на някоя от стените на фигурата (**border**).

### Примерен вход и изход



Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2		2		2		2	
3	outside	3	inside	2	border	6	border
10		1		2		0	

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2		15		15		15	
0	outside	13	outside	29	inside	37	outside
6		55		37		18	

### Насоки и подсказки

Примерна логика за решаване на задачата (не е единствената правилна):

- Може да разделим фигурата на **два правоъгълника** с обща стена (вж. чертежа).
- Една точка е **външна (outside)** за фигурата, когато е едновременно **извън** двета правоъгълника.
- Една точка е **вътрешна (inside)** за фигурата, ако е вътре в някой от правоъгълниците (изключвайки стените им) или лежи върху общата им стена.
- В **противен случай** точката лежи на стената на правоъгълника (**border**).

Примерен код (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```

h = ...
x = ...
y = ...

out_rectangle1 = ...
out_rectangle2 = ...

in_rectangle1 = ...
in_rectangle2 = ...

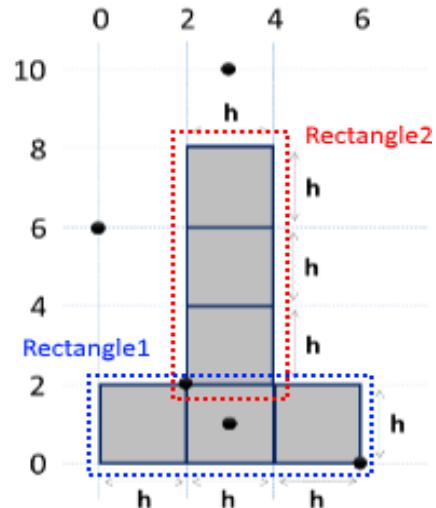
common_border = ...

if ...:
    print("outside")
elif ...:
    print("inside")
else:
    print("border")

```

### Тестване в Judge системата

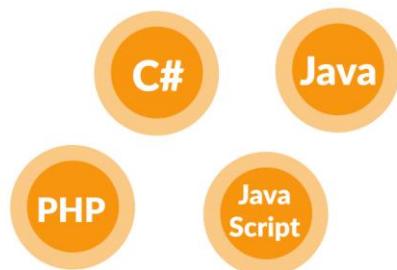
Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1051#12>.



Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 4.2. По-сложни проверки – изпитни задачи

В предходната глава се запознахме с **вложените условни конструкции** в езика **Python**. Чрез тях програмната логика в дадена програма може да бъде представена посредством **if конструкции**, които се съдържат една в друга. Разгледахме и по-сложния вариант **if-elif-else**, която позволява избор измежду редица от възможности. Следва да упражним и затвърдим наученото досега, като разгледаме няколко по-сложни задачи, давани на изпити. Преди да преминем към задачите, ще си припомним условните конструкции:

## Вложени проверки

```
if condition1:  
    if condition2:  
        # тяло  
    else:  
        # тяло
```



Запомнете, че не е добра практика да пишете **дълбоко вложени условни конструкции** (с ниво на влагане повече от три). Избягвайте влагане на повече от три условни конструкции една в друга. Това усложнява кода и затруднява неговото четене и разбиране.

## if-elif-else проверки

Когато работата на програмата ни зависи от стойността на една променлива, можем да правим последователни проверки с множество **if-elif-else** блокове:

```
if condition1:  
    # тяло  
elif condition2:  
    # тяло  
elif condition3:  
    # тяло  
else:  
    # тяло
```

Тялото може се състои от какъвто и да е код, стига да отговаря на синтактичните особености на езика и да е **вложен с една табуляция** навътре.

## Изпитни задачи

Сега, след като си припомнхме как се използват условни конструкции и **как се влагат една в друга условни конструкции** за реализиране на по-сложни проверки и програмна логика, нека решим няколко изпитни задачи.

## Задача: навреме за изпит

Студент трябва да отиде **на изпит в определен час** (например в 9:30 часа). Той идва в изпитната зала в даден **час на пристигане** (например 9:40). Счита се, че студентът е дошъл **навреме**, ако е пристигнал **в часа на изпита или до половин час преди това**. Ако е пристигнал **по-рано**, повече от 30 минути, той е **подранил**. Ако е дошъл **след часа на изпита**, той е **закъснял**.

Напишете програма, която въвежда време на изпит и време на пристигане и отпечатва дали студентът е дошъл **навреме**, дали е **подранил** или е **закъснял**, както и **с колко часа или минути** е подранил или закъснял.

### Входни данни

От конзолата се четат **четири цели числа** (по едно на ред):

- Първият ред съдържа **час на изпита** – цяло число от 0 до 23.
- Вторият ред съдържа **минута на изпита** – цяло число от 0 до 59.
- Третият ред съдържа **час на пристигане** – цяло число от 0 до 23.
- Четвъртият ред съдържа **минута на пристигане** – цяло число от 0 до 59.

### Изходни данни

На първия ред отпечатайте:

- "**Late**", ако студентът пристига **по-късно** от часа на изпита.
- "**On time**", ако студентът пристига **точно** в часа на изпита или до 30 минути по-рано.
- "**Early**", ако студентът пристига повече от 30 минути **преди** часа на изпита.

Ако студентът пристига с поне минута разлика от часа на изпита, отпечатайте на следващия ред:

- "**mm minutes before the start**" за идване по-рано с по-малко от час.
- "**hh:mm hours before the start**" за подраняване с 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:05".
- "**mm minutes after the start**" за закъснение под час.
- "**hh:mm hours after the start**" за закъснение от 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:03".

### Примерен вход и изход

Вход	Изход
9	
30	Late 20 minutes after the start
9	
50	

Вход	Изход
16	
00	Early 1:00 hours before the start
15	
00	

Вход	Изход
9	
00	On time 30 minutes before the start
8	
30	

Вход	Изход
9	
00	Late 1:30 hours after the start
10	
30	

Вход	Изход
14	
00	On time 5 minutes before the start
13	
55	

Вход	Изход
10	
00	On time
10	
00	

## Насоки и подсказки



Препоръчително е **да прочетете няколко пъти заданието** на дадена задача, като си водите записи и си скицирате примерите, докато разсъждавате над тях, преди да започнете писането на код.

## Обработка на входните данни

Съгласно заданието очакваме да ни бъдат подадени **четири** поредни реда с различни **цели числа**. Разглеждайки дадените входни параметри можем да се спрем на типа **int**, тъй като той удовлетворява очакваните ни стойности. Едновременно **четем** входа и **парсваме** стринговата стойност към избрания от нас тип данни за **цяло число**:

```
exam_hours = int(input())
exam_minutes = int(input())
arrival_hours = int(input())
arrival_minutes = int(input())
```

Разглеждайки очаквания изход, можем да създадем променливи, които да съдържат различните видове изходни данни, с цел да избегнем използването на т.нар. "magic strings" в кода:

```
late = "Late"
on_time = "On time"
early = "Early"
```

## Изчисления

След като прочетохме входа, можем да започнем да разписваме логиката за изчисление на резултата. Нека първо да изчислим **началния час** на изпита в **минути** за по-лесно и точно сравнение:

```
exam_time = (exam_hours * 60) + exam_minutes
```

Нека изчислим по същата логика и времето на пристигане на студента:

```
arrival_time = (arrival_hours * 60) + arrival_minutes
```

Остава ни да пресметнем разликата в двете времена, за да можем да определим кога и с какво време спрямо изпита е пристигнал студентът:

```
total_minutes_difference = arrival_time - exam_time
```

Следващата ни стъпка е да направим необходимите проверки и изчисления, като накрая ще изведем резултата от тях. Нека разделим изхода на **две** части:

- Първо да покажем кога е пристигнал студентът - дали е **подранил**, **закъснял** или е пристигнал **навреме**. За целта ще се спрем на **if-else** конструкция.
- След това ще покажем **времевата разлика**, ако студентът пристигне в **различно време** от началния час на изпита.

С цел да спестим една допълнителна проверка (**else**), можем по подразбиране да приемем, че студентът е закъснял.

След което, съгласно условието, проверяваме дали разликата във времената е **повече от 30 минути**. Ако това е така, приемаме, че е **подранил**. Ако не влезем в първото условие, то следва да проверим само дали **разликата е по-малка или равна на нула** ( $\leq 0$ ), с което проверяваме условието, студентът да е дошъл в рамките на от **0 до 30 минути** преди изпита.

При всички останали случаи приемаме, че студентът е **закъснял**, което сме направили по подразбиране, и не е нужна допълнителна проверка:

```
student_arrival = late
if total_minutes_difference < -30:
    student_arrival = early
elif total_minutes_difference <= 0:
    student_arrival = on_time
```

За финал ни остава да разберем и покажем с **каква разлика от времето на изпита е пристигнал**, както и дали тази разлика показва време на пристигане **преди или след изпита**.

Правим проверка дали разликата ни е **над един час**, за да изпишем съответно часове и минути в желания по задание **формат**, или е **под един час**, за да покажем **само минути** като формат и описание. Остава да направим още една проверка - дали времето на пристигане на студента е **преди** или **след** началото на изпита:

```
result = ""
if total_minutes_difference != 0:
    hours_difference = abs(total_minutes_difference) // 60
    minutes_difference = abs(total_minutes_difference) % 60
```

```

if hours_difference > 0:
    result = \
        f"{hours_difference}:{minutes_difference:02} hours"
else:
    result = f"{minutes_difference} minutes"

if total_minutes_difference < 0:
    result += " before the start"
else:
    result += " after the start"

```

## Отпечатване на резултата

Накрая остава да изведем резултата на конзолата. По задание, ако студентът е дошъл точно на време (без нито една минута разлика), не трябва да изваждаме втори резултат. Затова правим следната проверка:

```

print(student_arrival)
if result:
    print(result)

```

Реално за целите на задачата извеждането на резултата **на конзолата** може да бъде направен и в по-ранен етап - още при самите изчисления. Това като цяло не е много добра идея. **Защо?**

Нека разгледаме идеята, че кодът ни не е 10 реда, а 100 или 1000! Някой ден ще се наложи извеждането на резултата да не бъде в конзолата, а да бъде записан във **файл** или показан на **уеб приложение**. Тогава на колко места в кода ще трябва да бъдат нанесени корекции поради тази смяна? И дали няма да пропуснем някое място?



Винаги си мислете за кода с логическите изчисления, като за отделна част, различна от обработката на входните и изходните данни. Той трябва да може да работи без значение как му се подават данните и къде ще трябва да бъде показан резултатът.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1052#0>.

## Задача: пътешествие

Странно, но повечето хора си планиват от рано почивката. Млад програмист разполага с **определен бюджет** и свободно време в даден сезон.

Напишете програма, която да приема **на входа** бюджета и сезона, а **на изхода** да изкарва **къде ще почива** програмистът и **колко ще похарчи**.

Бюджетът определя дестинацията, а сезонът определя колко от бюджета ще бъде изхарчен. Ако е лято, ще почива на къмпинг, а зимата - в хотел. Ако е в Европа, независимо от сезона, ще почива в хотел. Всеки къмпинг или хотел, според дестинацията, има собствена цена, която отговаря на даден процент от бюджета:

- При **100 лв.** или по-малко – някъде в **България**.
  - Лято – 30% от бюджета.
  - Зима – 70% от бюджета.
- При **1000 лв.** или по малко – някъде на **Балканите**.
  - Лято – 40% от бюджета.
  - Зима – 80% от бюджета.
- При **повече от 1000 лв.** – някъде из **Европа**.
  - При пътуване из Европа, независимо от сезона, ще похарчи **90%** от бюджета.

## Входни данни

Входът се чете от конзолата и се състои от **два реда**:

- На **първия** ред получаваме **бюджета** - реално число в интервал [10.00 ... 5000.00].
- На **втория** ред – **един** от двата възможни сезона: "summer" или "winter".

## Изходни данни

На конзолата трябва да се отпечатат **два реда**.

- На **първи** ред – "Somewhere in {дестинация}" измежду "Bulgaria", "Balkans" и "Europe".
- На **втори** ред – "{Вид почивка} – {Похарчена сума}".
  - Почкивката може да е между "Camp" и "Hotel".
  - Сумата трябва да е закръглена с точност до втория десетичен символ.

## Примерен вход и изход

Вход	Изход
50 summer	Somewhere in Bulgaria Camp – 15.00

Вход	Изход
75 winter	Somewhere in Bulgaria Hotel – 52.50

Вход	Изход
312 summer	Somewhere in Balkans Camp – 124.80

Вход	Изход
1500 summer	Somewhere in Europe Hotel – 1350.00

## Насоки и подсказки

Типично, както и при другите задачи, можем да разделим решението на няколко части:

- Четене на входните данни
- Изчисления
- Отпечатване на резултата

## Обработка на входните данни

Прочитайки внимателно условието разбираме, че очакваме **два** реда с входни данни. Първият параметър е **реално число**, за което е добре да изберем подходящ тип на променливата. Ще се спрем на **float** като тип за бюджета, а за сезона - **string**:

```
budget = float(input())
season = input()
```



Винаги преценявайте какъв **тип стойност** се подава при входните данни, както и към какъв тип трябва да бъдат конвертираны тези данни, за да работят правилно създадените от вас програмни конструкции!

## Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи:

```
destination_result = ""
holiday_information = ""
money_spent = 0.00
```

Подобно на примера в предната задача, можем да инициализираме променливите с някои от изходните резултати - с цел спестяване на допълнително инициализиране.

Разглеждайки отново условието на задачата забелязваме, че основното разпределение за това къде ще почиваме се определя от **стойността на подадения бюджет**, т.е. основната ни логика се разделя на два случая:

- Ако бюджетът е **по-малък** от дадена стойност.
- Ако е **по-малък** от друга стойност, или е **повече** от дадена гранична стойност.

Спрямо това как си подредим логическата схема (в какъв ред ще обхождаме граничните стойности), ще имаме повече или по-малко проверки в условията. **Помислете защо!**

След това е необходимо да направим проверка за стойността на **подадения сезон**. Спрямо нея ще определим какъв процент от бюджета ще бъде похарчен, както и

Къде ще почива програмистът - в хотел или на къмпинг. Пример за един от възможните подходи за решение е:

```
if budget <= 100:
    destination_result = "Bulgaria"
    if season == "summer":
        money_spent = 0.30 * budget
        holiday_information = f"Camp - {money_spent:.2f}"
    else:
        money_spent = 0.70 * budget
        holiday_information = f"Hotel - {money_spent:.2f}"
elif budget <= 1000:
    destination_result = "Balkans"
    if season == "summer":
        money_spent = 0.40 * budget
        holiday_information = f"Camp - {money_spent:.2f}"
    else:
        money_spent = 0.80 * budget
        holiday_information = f"Hotel - {money_spent:.2f}"
else:
    destination_result = "Europe"
    money_spent = 0.90 * budget
    holiday_information = f"Hotel - {money_spent:.2f}"
```

Винаги можем да инициализираме дадена стойност на параметъра и след това да направим само една проверка. Това ни спестява една логическа стъпка. Например следният блок:

```
if budget <= 100:
    destination_result = "Bulgaria"
    if season == "summer":
        money_spent = 0.30 * budget
        holiday_information = f"Camp - {money_spent:.2f}"
    else:
        money_spent = 0.70 * budget
        holiday_information = f"Hotel - {money_spent:.2f}"
```

може да бъде съкратен до този вид:

```
destination_result = "Bulgaria"
money_spent = 0.70 * budget
holiday_information = f"Hotel - {money_spent:.2f}"
if season == "summer":
    money_spent = 0.30 * budget
    holiday_information = f"Camp - {money_spent:.2f}"
```

## Отпечатване на резултата

Остава ни да покажем изчисления резултат на конзолата:

```
print("Somewhere in " + destination_result)
print(holiday_information)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1052#1>.

## Задача: операции между числа

Напишете програма, която чете **две цели числа (n1 и n2)** и **оператор**, с който да се извърши дадена математическа операция с тях. Възможните операции са: **събиране (+)**, **изваждане (-)**, **умножение (\*)**, **деление (/)** и **модулно деление (%)**. При събиране, изваждане и умножение на конзолата трябва да се отпечата резултата и дали той е **четен** или **нечетен**. При обикновено деление – **единствено резултата**, а при модулно деление – **остатъка**. Трябва да се има предвид, че **делителят може да е равен на нула (= 0)**, а на нула не се дели. В този случай трябва да се отпечата специално съобщение.

### Входни данни

От конзолата се прочитат 3 реда:

- N1 – цяло число в интервала [0 ... 40 000].
- N2 – цяло число в интервала [0 ... 40 000].
- Оператор – един символ измежду: "+", "-", "\*", "/", "%".

### Изходни данни

Да се отпечата на конзолата **един ред**:

- Ако операцията е **събиране**, **изваждане** или **умножение**:
  - " $\{N1\} \{\text{оператор}\} \{N2\} = \{\text{резултат}\}$ " –  $\{\text{even/odd}\}$ ".
- Ако операцията е **деление**:
  - " $\{N1\} / \{N2\} = \{\text{резултат}\}$ " – резултатът е **форматиран до втория символ след десетичния знак**.
- Ако операцията е **модулно деление**:
  - " $\{N1\} \% \{N2\} = \{\text{остатък}\}$ ".
- В случай на **деление на 0 (нула)**:
  - "Cannot divide  $\{N1\}$  by zero".

### Примерен вход и изход

Вход	Изход
10	
1	$10 - 1 = 9$ – odd
-	

Вход	Изход
7	
3	$7 * 3 = 21$ – odd
*	

Вход	Изход
123	
12	$123 / 12 = 10.25$
/	

Вход	Изход
10	
3	$10 \% 3 = 1$
%	

Вход	Изход
10	
12	$10 + 12 = 22$ – even
+	

Вход	Изход
112	
0	Cannot divide 112 by zero
/	

## Насоки и подсказки

Задачата не е сложна, но има доста редове код за писане.

### Обработка на входните данни

След прочитане на условието разбираме, че очакваме три реда с входни данни. На първите два реда ни се подават **цели числа** (в указания от заданието диапазон), а на третия – **аритметичен символ**:

```
n1 = int(input())
n2 = int(input())
n_operator = input()
```

### Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи. В едната ще пазим **резултата от изчисленията**, а другата ще използваме за **крайния изход** на програмата:

```
result = 0
output = ""
```

Прочитайки внимателно условието разбираме, че има случаи, в които не трябва да правим **никакви** изчисления, а просто да изведем резултат. Следователно първо може да проверим дали второто число е **0** (нула), както и дали операцията е **деление** или **модулно деление**, след което да инициализираме резултата:

```
if n2 == 0 and (n_operator == "/" or n_operator == "%"):
    output = f"Cannot divide {n1} by zero"
```

Нека сложим резултата като стойност при инициализацията на **output** параметъра. По този начин може да направим само **една проверка** – дали е необходимо да **преизчислим** и **заменим** този резултат.

Спрямо това кой подход изберем, следващата ни проверка ще бъде или обикновен **elif** или единичен **if**. В тялото на тази проверка, с допълнителни проверки за начина на изчисление на резултата спрямо подадения оператор, можем да разделим логиката спрямо **структурата** на очаквания **резултат**.

От условието можем да видим, че за **събиране (+)**, **изважддане (-)** или **умножение (\*)** очакваният резултат има еднаква структура: "[n1] {оператор} {n2} = {результат} – {even/odd}", докато за **деление (/)** и за **модулно деление (%)** резултатът има различна структура:

```
elif n_operator == "/":
    result = n1 / n2
    output = f"{n1} {n_operator} {n2} = {result:.2f}"
elif n_operator == "%":
    result = n1 % n2
    output = f"{n1} {n_operator} {n2} = {result}"
```

Завършваме с проверките за събиране, изважддане и умножение:

```
else:
    if n_operator == "+":
        result = n1 + n2
    elif n_operator == "-":
        result = n1 - n2
    elif n_operator == "*":
        result = n1 * n2

    output = f"{n1} {n_operator} {n2} = {result} " \
             f"- {('even' if result % 2 == 0 else 'odd')} }"
```

При кратки и ясни проверки, както в горния пример за четно и нечетно число, е възможно да се използва **тернарен оператор**, който просто спестява няколко реда код. Нека разгледаме възможната проверка **с и без** тернарен оператор.

**Без използване на тернарен оператор** кодът е по-дълъг, но се чете лесно:

```
number_is = ""
if result % 2 == 0:
    number_is = "even"
else:
    number_is = "odd"
```

**С използване на тернарен оператор** кодът е много по-кратък, но може да изисква допълнителни усилия, за да бъде прочетен и разбран като логика:

```
number_is = 'even' if result % 2 == 0 else 'odd'
```

## Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата:

```
print(output)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1052#2>.

## Задача: билети за мач

Група запалянковци решили да си закупят **билети за Евро 2016**. Цената на билета се определя спрямо **две** категории:

- VIP – **499.99** лева.
- Normal – **249.99** лева.

Запалянковците имат определен бюджет, а броят на хората в групата определя какъв процент от бюджета трябва да се задели за транспорт:

- От 1 до 4 – 75% от бюджета.
- От 5 до 9 – 60% от бюджета.
- От 10 до 24 – 50% от бюджета.
- От 25 до 49 – 40% от бюджета.
- 50 или повече – 25% от бюджета.

Напишете програма, която да **пресмята дали с останалите пари от бюджета могат да си купят билети за избраната категория**, както и колко пари ще им останат или ще са им нужни.

## Входни данни

Входът се чете от конзолата и съдържа **точно 3 реда**:

- На **първия** ред е **бюджетът** – реално число в интервала [1 000.00 ... 1 000 000.00].
- На **втория** ред е **категорията** – "VIP" или "Normal".
- На **третия** ред е **броят на хората в групата** – цяло число в интервала [1 ... 200].

## Изходни данни

Да се **отпечата** на конзолата **един ред**:

- Ако бюджетът е достатъчен:
  - "Yes! You have {N} leva left." – където N са останалите пари на групата.

- Ако бюджетът НЕ Е достатъчен:

- "Not enough money! You need {M} leva." – където M е сумата, която не достига.

Сумите трябва да са форматирани с точност до две цифри след десетичния знак.

### Примерен вход и изход

Вход	Изход	Обяснения
1000 Normal 1	Yes! You have 0.01 leva left.	1 човек : 75% от бюджета отиват за транспорт. Остават: $1000 - 750 = 250$ . Категория Normal: билетът струва $249.99 * 1 = 249.99$ $249.99 < 250$ : остават му $250 - 249.99 = 0.01$

Вход	Изход	Обяснения
30000 VIP 49	Not enough money! You need 6499.51 leva.	49 човека: 40% от бюджета отиват за транспорт. Остават: $30000 - 12000 = 18000$ . Категория VIP: билетът струва $499.99 * 49 = 24499.51$ $24499.51 < 18000$ . Не стигат $24499.51 - 18000 = 6499.51$

### Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията, описани в условието на задачата, за да проверим дали ще стигнат парите.

### Обработка на входните данни

Нека прочетем внимателно условието и да разгледаме какво се очаква да получим като **входни данни**, какво се очаква да **върнем като резултат**, както и кои са **основните стъпки** при разбиването на логическата схема. Като начало, нека обработим и запазим входните данни в **подходящи** за това **променливи**:

```
budget = float(input())
ticket_type = input()
people = int(input())
```

### Изчисления

Нека създадем и инициализираме нужните за изчисленията променливи:

```
transport_charges = 0.00
money_for_tickets = 0.00
money_difference = 0.00
```

Нека отново прегледаме условието. Трябва да направим **две** различни блок изчисления. От първите изчисления трябва да разберем каква част от бюджета ще трябва да заделим за **транспорт**. За логиката на тези изчисления забелязваме, че има значение единствено **броят на хората в групата**. Следователно ще направим логическата разбивка спрямо броя на запалянковците. Ще използваме условна конструкция - поредица от **if-elif** блокове:

```
if people <= 4:
    transport_charges = 0.75 * budget
elif people <= 9:
    transport_charges = 0.60 * budget
elif people <= 24:
    transport_charges = 0.50 * budget
elif people <= 49:
    transport_charges = 0.40 * budget
elif people >= 50:
    transport_charges = 0.25 * budget
```

От вторите изчисления трябва да намерим каква сума ще ни е необходима за закупуване на **билети за групата**. Според условието, това зависи единствено от типа на билетите, които трябва да закупим. Нека използваме **if-elif** условна конструкция:

```
if ticket_type == "Normal":
    money_for_tickets = people * 249.99
elif ticket_type == "VIP":
    money_for_tickets = people * 499.99
```

След като сме изчислили какви са **транспортните разходи и разходите за билети**, ни остава да изчислим крайния резултат и да разберем **ще успее** ли групата от запалянковци да отиде на Евро 2016 или **няма да успее** при така подадените параметри.

За извеждането на резултата, за да си спестим една **проверка** в конструкцията, приемаме, че групата по подразбиране ще може да отиде на Евро 2016:

```
money_difference = \
    budget - transport_charges - money_for_tickets

result = f"Yes! You have {money_difference:.2f} leva left."
if money_difference < 0:
    result = "Not enough money! " \
        f"You need {abs(money_difference):.2f} leva."
```

## Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1052#3>.

### Задача: хотелска стая

Хотел предлага два вида стаи: студио и апартамент.

Напишете програма, която изчислява цената за целия престой за студио и апартамент. Цените зависят от месеца на престоя:

Май и октомври	Юни и септември	Юли и август
Студио – 50 лв./нощувка	Студио – 75.20 лв./нощувка	Студио – 76 лв./нощувка
Апартамент – 65 лв./нощувка	Апартамент – 68.70 лв./нощувка	Апартамент – 77 лв./нощувка

Предлагат се и следните отстъпки:

- За студио, при повече от 7 нощувки през май и октомври: 5% намаление.
- За студио, при повече от 14 нощувки през май и октомври: 30% намаление.
- За студио, при повече от 14 нощувки през юни и септември: 20% намаление.
- За апартамент, при повече от 14 нощувки, без значение от месеца: 10% намаление.

### Входни данни

Входът се чете от конзолата и съдържа точно два реда:

- На първия ред е месецът – May, June, July, August, September или October.
- На втория ред е броят на нощувките – цяло число в интервала [0 ... 200].

### Изходни данни

Да се отпечатат на конзолата два реда:

- На първия ред: "Apartment: { цена за целият престой } lv".
- На втория ред: "Studio: { цена за целият престой } lv".

Цената за целия престой да е форматирана с точност до два символа след десетичния знак.

### Примерен вход и изход

Вход	Изход
June 14	Apartment: 961.80 lv. Studio: 1052.80 lv

Вход	Изход
August 20	Apartment: 1386.00 lv. Studio: 1520.00 lv.

Вход	Изход	Обяснения
May 15	Apartment: 877.50 lv. Studio: 525.00 lv.	През <b>май</b> , при повече от <b>14 нощувки</b> , намаляваме цената на <b>студиото с 30%</b> ( $50 - 15 = 35$ ), а на <b>апартамента – с 10%</b> ( $65 - 6.5 = 58.5$ ). Целият престой в <b>апартамент – 877.50 лв.</b> Целият престой в <b>студио – 525.00 лв.</b>

## Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията според описания ценоразпис и правилата за отстъпките и накрая ще отпечатаме резултата.

### Обработка на входните данни

Съгласно условието на задачата очакваме да получим два реда входни данни - на първия ред **месец**, през който се планива престой, а на втория - **броя нощувки**. Нека обработим и запазим входните данни в подходящи за това променливи:

```
month = input()
nights = int(input())
```

### Изчисления

След това да създадем и инициализираме нужните за изчисленията променливи:

```
studio_price = 50.00
apartment_price = 65.00
studio_rent = 0.00
apartment_rent = 0.00
```

Разглеждайки отново условието забелязваме, че основната ни логика зависи от това какъв **месец** ни се подава, както и от броя на **нощувките**.

Като цяло има различни подходи и начини да се направят въпросните проверки, но нека се спрем на основна условна конструкция **if-elif**, като в различните случаи ще използваме съответно вложени условни конструкции **if** и **if-elif**.

Нека започнем с първата група месеци: **Май** и **Октомври**. За тези два месеца цената на престой е **еднаква** и за двета типа настаняване - в **студио** и в **апартамент**. Съответно остава само да направим вътрешна проверка спрямо **броят нощувки**, за да преизчислим **съответната цена** (ако се налага):

```

if month == "May" or month == "October":
    studio_price = 50.00
    apartment_price = 65.00

    studio_rent = studio_price * nights
    apartment_rent = apartment_price * nights
    if nights > 14:
        studio_rent *= 0.70
        apartment_rent *= 0.90
    elif nights > 7:
        studio_rent *= 0.95

```

За следващите месеци логиката и изчисленията ще са донякъде идентични:

```

elif month == "June" or month == "September":
    studio_price = 75.20
    apartment_price = 68.70

    studio_rent = studio_price * nights
    apartment_rent = apartment_price * nights

    if nights > 14:
        studio_rent *= 0.80
        apartment_rent *= 0.90
elif month == "July" or month == "August":
    studio_price = 76.00
    apartment_price = 77.00

    studio_rent = studio_price * nights
    apartment_rent = apartment_price * nights
    if nights > 14:
        apartment_rent *= 0.90

```

След като изчислихме какви са съответните цени и крайната сума за престоя – нека да си изведем във форматиран вид резултата, като преди това го запишем в изходните ни променливи – **studio\_info** и **apartment\_info**:

```

studio_info = f"Studio: {studio_rent:.2f} lv."
apartment_info = f"Apartment: {apartment_rent:.2f} lv."

```

За изчисленията на изходните параметри използваме **форматирането .2f**. Този начин на форматиране **закръгля десетично** число до **зададен брой цифри** (в случая 2) след десетичния знак, а на целите числа просто добавя необходимия брой символи след десетичната запетая.

## Отпечатване на резултата

На края остава да покажем изчислените резултати на конзолата.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1052#4>.

# Глава 5.1. Повторения (цикли)

В настоящата глава ще се запознаем с конструкциите за **повторение** на група **команди**, известни в програмирането с понятието "цикли". Ще напишем няколко цикъла с използване на оператора **for** в класическата му форма. Накрая ще решим няколко практически задачи, изискващи повторение на поредица от действия, като използваме цикли.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=pvTMTGsooMk>.

### Повторения на блокове код (for цикъл)

В програмирането често пъти се налага да изпълним блок с команди няколко пъти. За целта се използват т.нр. **цикли**. Нека разгледаме един пример за **for** цикъл, който преминава последователно през числата от 1 до 10 и ги отпечатва:

```
for i in range(1, 11):
    print('i = ' + str(i))
```

Цикълът започва с **оператора for** и преминава през всички стойности за дадена променлива в даден интервал, например всички числа от 1 до 10 включително (без да включва 11), и за всяка стойност изпълнява поредица от команди.

В декларацията на цикъла може да се зададе **начална стойност** и **краяна стойност**, като крайната стойност не е включена в диапазона. **Тялото на цикъла** представлява блок с една или няколко команди. На фигурата по-долу е показана структурата на един **for** цикъл:



В повечето случаи един **for** цикъл се завърта от **1** до **n** (например от 1 до 10). Целта на цикъла е да се премине последователно през числата 1, 2, 3, ..., n и за всяко от тях да се изпълни някакво действие. В примера по-горе променливата **i** приема стойности от 1 до 10 и в тялото на цикъла се отпечатва текущата стойност. Цикълът се повтаря 10 пъти и всяко от тези повторения се нарича "итерация".

#### Пример: числа от 1 до 100

Да се напише програма, която **печата числата от 1 до 100**. Програмата не приема вход и отпечатва числата от 1 до 100 едно след друго, по едно на ред.

## Насоки и подсказки

Можем да решим задачата с **for цикъл**, с който преминаваме с променливата **i** през числата от 1 до 100 и ги печатаме в тялото на цикъла:

```
for i in range(1, 101):
    print('i = ' + str(i))
```

Стартираме програмата с [Ctrl+Shift+F10] или с десен бутон + [Run], и я **тестваме**:

```
Run: numbers_1_to_100 ×
C:\Projects\Loops\
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#0>.

Трябва да получите **100 точки** (напълно коректно решение).

## Пример: числа до 1000, завършващи на 7

Да се напише програма, която намира всички числа в интервала [1 ... 1000], които завършват на 7.

## Насоки и подсказки

Задачата можем да решим като комбинираме **for цикъл** за преминаваме през числата от 1 до 1000 и **проверка** за всяко число дали завършва на 7. Има и други решения, разбира се, но нека решим задачата чрез **завъртане на цикъл + проверка**:

```
for i in range(1, 1000):
    if i % 10 == 7:
        print(str(i))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#1>.

## Пример: всички латински букви

Да се напише програма, която отпечатва буквите от латинската азбука: **a, b, ..., z**.

## Насоки и подсказки

Полезно е да се знае, че **for** циклите не работят само с числа. Може да решим задачата като завъртим **for** цикъл, който преминава последователно през всички букви от латинската азбука:

```
for i in range(ord('a'), ord('z') + 1):
    print(chr(i))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#2>.

## Пример: сумиране на числа

Да се напише програма, която въвежда **n** цели числа и ги сумира.

- От първия ред на входа се въвежда броят числа **n**.
- От следващите **n** реда се въвежда по едно число.
- Числата се сумират и накрая се отпечатва резултатът.

## Примерен вход и изход

Вход	Изход
2	
10	
20	30

Вход	Изход
3	
-10	
-20	
-30	-60

Вход	Изход
4	
45	
-20	
7	
11	43

Вход	Изход
1 999	999

Вход	Изход
0	0

## Насоки и подсказки

Можем да решим задачата за сумиране на числа по следния начин:

- Четем входното число **n** от конзолата.
- Започваме първоначално със сума **sum = 0**.
- Въртим цикъл от 0 до **n**. На всяка стъпка от цикъла четем число **num** и го добавяме към сумата **sum**.
- Накрая отпечатваме получената сума **sum**.

Ето и сорс кода на решението:

```

n = int(input())
sum_nums = 0

for i in range(0, n):
    current_num = int(input())
    sum_nums = sum_nums + current_num

print("sum = " + str(sum_nums))

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#3>.

## Пример: най-голямо число

Да се напише програма, която въвежда **n** цели числа ( $n > 0$ ) и намира най-голямото измежду тях. На първия ред на входа се въвежда броят числа **n**. След това се въвеждат самите числа, по едно на ред. Примери:

### Примерен вход и изход

Вход	Изход
2	
100	
99	100

Вход	Изход
3	
-10	
20	
-30	20

Вход	Изход
4	
45	
-20	
7	99
99	

Вход	Изход
1 999	999

Вход	Изход
2	
-1	
-2	-1

## Насоки и подсказки

Първо въвеждаме едно число **n** (броят числа, които предстои да бъдат въведени). Задаваме на текущия максимум **max** първоначална неутрална стойност, например **-1000000000000000**. С помощта на **for** цикъл, чрез който итерираме **n-1** пъти, прочитаме по едно цяло число **num**. Ако прочетеното число **num** е по-голямо от текущия максимум **max**, присвояваме стойността на **num** в променливата **max**. Накрая, в **max** трябва да се е запазило най-голямото число. Отпечатваме го на конзолата. Ето и цялостна имплементация на описания алгоритъм:

```

n = int(input('n = '))
max_num = -1000000000000000

for i in range(n):
    num = int(input())
    if num > max_num:
        max_num = num

print('max = ' + str(max_num))

```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#4>.

### Пример: най-малко число

Да се напише програма, която въвежда **n** цели числа ( $n > 0$ ) и намира **най-малкото** измежду тях. На първия ред е числото **n**, след него още **n** числа по едно на ред.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2 100 99	99	3 -10 20 -30	-30	4 45 -20 7 99	-20

#### Насоки и подсказки

Задачата е абсолютно аналогична с предходната, само че започване с друга неутрална начална стойност:

```

n = int(input('n = '))
min_num = 1000000000000000

# Use similar logic to previous problem

```

```
print('min = ' + str(min_num))
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#5>.

## Пример: лява и дясна сума

Да се напише програма, която въвежда  $2 * n$  цели числа и проверява дали **сумата на първите  $n$  числа** (лява сума) е равна на **сумата на вторите  $n$  числа** (дясна сума). При равенство се печата "Yes" + **сумата**, иначе се печата "No" + **разликата**. Разликата се изчислява като положително число (по абсолютна стойност). Форматът на изхода трябва да е като в примерите по-долу.

### Примерен вход и изход

Вход	Изход	Вход	Изход
2		2	
10		90	
90	Yes, sum = 100	9	No, diff = 1
60		50	
40		50	

### Насоки и подсказки

Първо въвеждаме числото  $n$ , след това първите  $n$  числа (**лявата** половина) и ги сумираме. Продължаваме с въвеждането на още  $n$  числа (**дясната** половина) и намираме и тяхната сума. Изчисляваме **разликата** между намерените суми по абсолютна стойност: **math.fabs(rightSum - leftSum)**. Ако разликата е 0, отпечатваме "Yes" + **сумата**, в противен случай - отпечатваме "No" + **разликата**:

```
import math

n = int(input())
left_sum = 0
right_sum = 0

for i in range(0, n):
    numbersForLeftSide = float(input())
    left_sum += numbersForLeftSide

# Read the right part of the numbers and sum them
# [REDACTED]

if left_sum == right_sum:
    print("Yes, sum = %d" % left_sum)
else:
    print("No, diff = %d" % math.fabs(right_sum - left_sum))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#6>.

### Пример: четна / нечетна сума

Да се напише програма, която въвежда **n** цели числа и проверява дали **сумата на числата на четни позиции** е равна на **сумата на числата на нечетни позиции**. При равенство печата "Yes" + **сумата**, иначе печата "No" + **разликата**. Разликата се изчислява по абсолютна стойност. Форматът на изхода трябва да е като в примерите по-долу.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
4		4		3	
10		3		5	No
50	Yes Sum = 70	5	Diff = 1	8	
60		1		1	
20		-2			

#### Насоки и подсказки

Въвеждаме числата едно по едно и изчисляваме двете **суми** (на числата на **четни** позиции и на числата на **нечетни** позиции). Както в предходната задача, изчисляваме абсолютната стойност на разликата и отпечатваме резултата ("Yes" + **сумата** при разлика 0 или "No" + **разликата** в противен случай):

```
import math

n = int(input())
even_sum = 0
odd_sum = 0

for i in range(1, n+1):
    num = int(input())
    if i % 2 == 0:
        even_sum += num
    else:
        odd_sum += num

# Print sum or difference
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#7>.

## Пример: сумиране на гласните букви

Да се напише програма, която въвежда **текст** (стринг), изчислява и отпечатва **сумата от стойностите на гласните букви** според таблицата по-долу:

a	e	i	o	u
1	2	3	4	5

### Примерен вход и изход

Вход	Изход
hello	6 (e+o = 2+4 = 6)
hi	3 (i = 3)

Вход	Изход
bamboo	9 (a+o+o = 1+4+4 = 9)
beer	4 (e+e = 2+2 = 4)

### Насоки и подсказки

Прочитаме входния текст **line**, зануяваме сумата и завъртаме цикъл, който преминава през всеки символ от входния текст. Проверяваме всяка буква **c** дали е гласна и съответно добавяме към сумата стойността ѝ:

```
line = input().lower()
sum_vowels = 0

for c in line:
    if c == 'a':
        sum_vowels += 1
    elif c == 'e':
        sum_vowels += 2
    elif c == 'i':
        sum_vowels += 3
    elif c == 'o':
        sum_vowels += 4
    elif c == 'u':
        sum_vowels += 5

print(sum_vowels)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#8>.

## Какво научихме от тази глава?

Можем да повтаряме блок код с **for** цикъл:

```
for i in range(1, 11):
    print('i = ' + str(i))
```

Можем да четем поредица от **n** числа от конзолата:

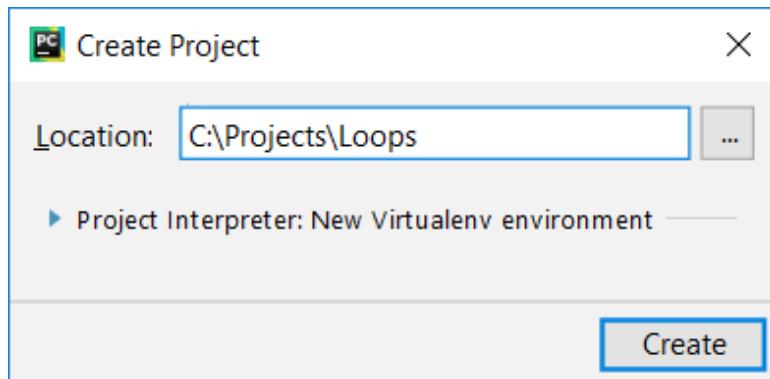
```
for i in range(n):
    num = int(input())
```

## Упражнения: повторения (цикли)

След като се запознахме с циклите, идва време да затвърдим знанията си на практика, а както знаете, това става с много писане на код. Да решим няколко задачи за упражнение.

### Създаване на нов проект в PyCharm

Създаваме нов проект в PyCharm (от [File] -> [New Project]), за да организираме по-добре задачите за упражнение. Целта на този проект е да съдържа по един Python файл за всяка задача от упражненията:



### Задача: елемент, равен на сумата на останалите

Да се напише програма, която въвежда **n** цели числа и проверява дали сред тях съществува число, което е равно на сумата на всички останали. Ако има такъв елемент, се отпечатва "Yes" + неговата стойност, в противен случай - "No" + разликата между най-големия елемент и сумата на останалите (по абсолютна стойност).

### Примерен вход и изход

Вход	Изход	Коментар	Вход	Изход	Коментар
7 3 4 1 1 2 12 1	Yes Sum = 12	$3 + 4 + 1 + 2 + 1 + 1 = 12$	3 1 1 10	No Diff = 8	$ 10 - (1 + 1)  = 8$

### Насоки и подсказки

Трябва да изчислим **сумата** на всички елементи, да намерим **най-големия** от тях и да проверим търсеното условие.

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1053#9>.

### Задача: четни / нечетни позиции

Напишете програма, която чете  $n$  числа и пресмята **сумата**, **минимума** и **максимума** на числата на **четни** и **нечетни** позиции (броим от 1). Когато няма **минимален** / **максимален** елемент, отпечатайте "No".

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6 2 3 5 4 2 1	OddSum=9, OddMin=2, OddMax=5, EvenSum=8, EvenMin=1, EvenMax=4	2 1.5 -2.5	OddSum=1.5, OddMin=1.5, OddMax=1.5, EvenSum=-2.5, EvenMin=-2.5, EvenMax=-2.5	1 1	OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin=No, EvenMax=No

Вход	Изход	Вход	Изход	Вход	Изход
3 -1 -2 -3	OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2	1 -5	OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin=No, EvenMax=No	5 3 -2 8 11 -3	OddSum=8, OddMin=-3, OddMax=8, EvenSum=9, EvenMin=-2, EvenMax=11

## Насоки и подсказки

Задачата обединява няколко предходни задачи: намиране на **минимум**, **максимум** и **сума**, както и обработка на елементите от **четни** и **нечетни позиции**. Припомнете си ги.

В тази задача е по-добре да се работи с **дробни числа** (не цели). Сумата, минимумът и максимумът също са дробни числа. Трябва да използваме **нейтрална начална стойност** при намиране на минимум / максимум, например **1000000000.0** и **-1000000000.0**. Ако получим накрая неутралната стойност, печатаме "No".

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1053#10>.

## Задача: еднакви двойки

Дадени са  $2 * n$  числа. Първото и второто формират **двойка**, третото и четвъртото също и т.н. Всяка двойка има **стойност** – сумата от съставящите я числа. Напишете програма, която проверява **дали всички двойки имат еднаква стойност**. В случай, че е еднаква отпечатайте "Yes, value=..." + **стойността**, в противен случай отпечатайте **максималната разлика** между две последователни двойки в следния формат - "No, maxdiff=..." + **максималната разлика**. Входът се състои от число  $n$ , следвано от  $2*n$  цели числа, всички по едно на ред.

### Примерен вход и изход

Вход	Изход	Коментар	Вход	Изход	Коментар
3 1 2 0 3 4 -1	Yes, value=3	стойности = {3, 3, 3} еднакви стойности	2 1 2 2 2	No, maxdiff=1	стойности = {3, 4} разлики = {1} макс. разлика = 1

Вход	Изход	Коментар	Вход	Изход	Коментар
2 -1 2 0 -1	No, maxdiff=2	стойности = {1, -1} разлики = {2} макс. разлика = 2	1 5 5	Yes, value=10	стойности = {10} една стойност → еднакви стойности

## Насоки и подсказки

Прочитаме входните числа **по двойки**. За всяка двойка пресмятаме **сумата** ѝ. Докато четем входните двойки, за всяка двойка, без първата, трябва да пресметнем **разликата с предходната**. За целта е необходимо да пазим в отделна променлива сумата на предходната двойка. Накрая намираме **най-голямата разлика** между две двойки. Ако е 0, печатаме "Yes" + стойността, в противен случай - "No" + разликата.

## Тестване в Judge системата

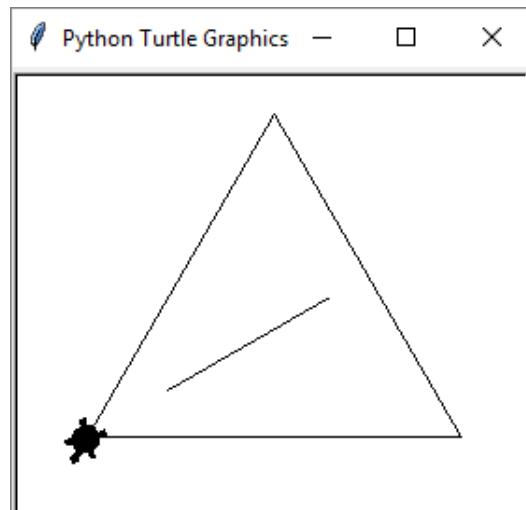
Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1053#11>.

## Упражнения: графични и уеб приложения

В настоящата глава се запознахме с **циклите** като конструкция в програмирането, която ни позволява да повтаряме многоократно дадено действие или група от действия. Сега нека си поиграем с тях. За целта ще начертаем няколко фигури, които се състоят от голям брой повтарящи се графични елементи, но този път не на конзолата, а в графична среда, използвайки "**графика с костенурка**". Ще е интересно. И никак не е сложно. Опитайте!

## Задача: чертане с костенурка – графично GUI приложение

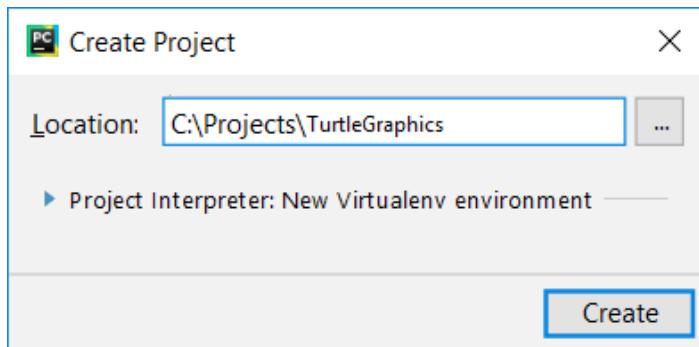
Целта на следващото упражнение е да си поиграем с една **библиотека за рисуване**, известна като "**графика с костенурка**" (*turtle graphics*). Ще изградим графично приложение, в което ще **рисуваме различни фигури**, придвижвайки нашата "**костенурка**" по екрана чрез операции от типа "отиди напред 100 позиции", "завърти се надясно на 30 градуса", "отиди напред още 50 позиции". Приложението ще изглежда приблизително така:



Нека първо се запознаем с **концепцията за рисуване "Turtle Graphics"**. Може да разгледаме следните източници:

- Дефиниция на понятието "turtle graphics" (концепцията за движение и ротация): <http://c2.com/cgi/wiki?TurtleGraphics>
- Статия за "turtle graphics" в Wikipedia (с повече примери за интересни графики) – [https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)
- Интерактивен онлайн инструмент за чертане с костенурка – <https://blockly-games.appspot.com/turtle>

Започваме, като създаваме нов проект с PyCharm:



В новосъздадения проект добавяме нов **Python File**. За чертането ще използваме външната библиотека **turtle**. Тя дефинира клас **Turtle**, който представлява **костенурка за рисуване**. За да я използваме, добавяме следния код, най-отгоре в началото на Python файла:

```
import turtle

my_turtle = turtle.Turtle()
```

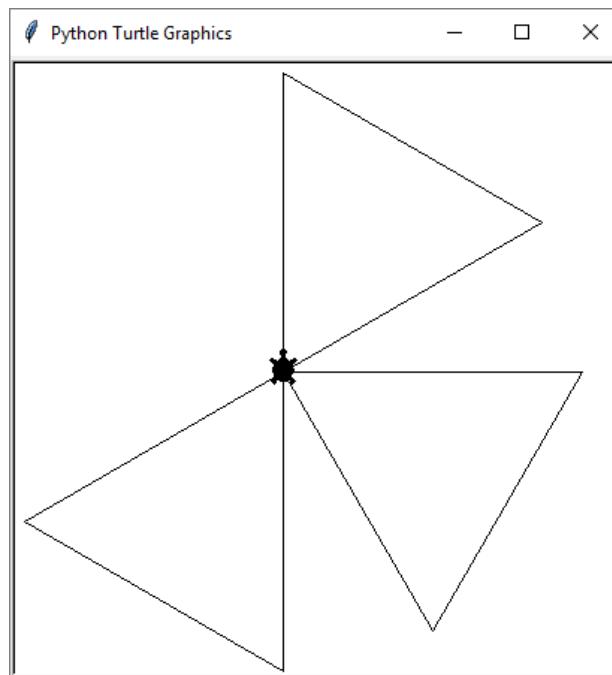
След това, за чертането, добавяме следния код:

```
# Change the shape of the turtle
my_turtle.shape("turtle")
```

```
for i in range(0, 3):
    # Draw a equilateral triangle
    my_turtle.left(30)
    my_turtle.forward(200)
    my_turtle.left(120)
    my_turtle.forward(200)
    my_turtle.left(120)
    my_turtle.forward(200)

turtle.done()
```

Горния код мести и върти костенурката, която в началото е в центъра на екрана (в средата на формата), и чертае равностранен триъгълник. Може да го редактирате и да си поиграете с него. **Стартираме** приложението:



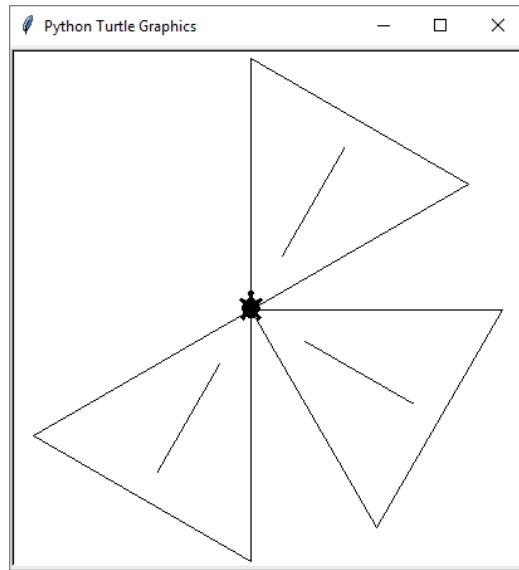
Сега може да променим и усложним горния код, за да получим по-сложна фигура:

```
for i in range(0, 3):
    # Draw a equilateral triangle
    my_turtle.left(30)
    my_turtle.forward(200)
    my_turtle.left(120)
    my_turtle.forward(200)
    my_turtle.left(120)
    my_turtle.forward(200)

    # Draw a line in the triangle
    my_turtle.left(-30)
    my_turtle.penup()
    my_turtle.backward(50)
    my_turtle.pendown()
    my_turtle.backward(100)
    my_turtle.penup()
    my_turtle.forward(150)
    my_turtle.pendown()
    my_turtle.left(30)
```

```
turtle.done()
```

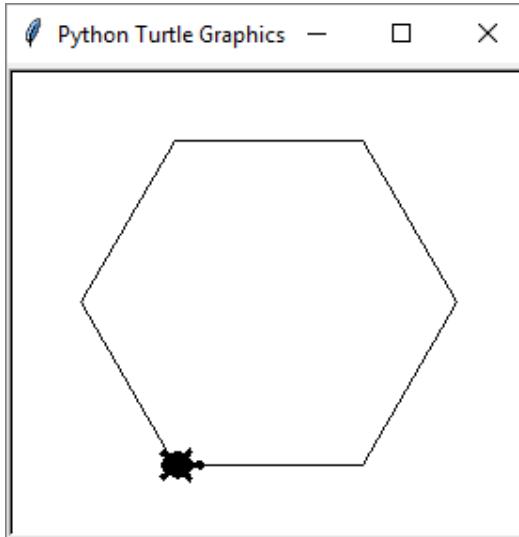
Отново **стартираме** приложението, за да видим резултата:



Вече нашата костенурката чертае по-сложни фигури чрез приятно анимирано движение.

### **Задача: \* чертане на шестоъгълник с костенурката**

Добавете нов Python файл, с примерно име `hexagon`, който ще чертае правилен шестоъгълник:



**Подсказка:**

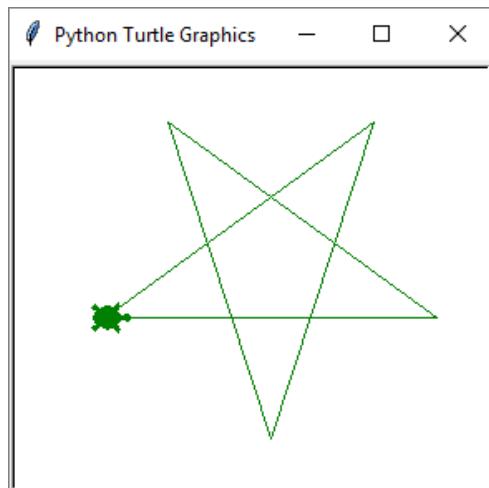
В цикъл повторете 6 пъти следното:

- Ротация на 60 градуса.

- Движение напред 100.

### Задача: \* чертане на звезда с костенурката

Добавете нов Python файл `star.py`, който чертае звезда с 5 върха (петолъчка), като на фигурата по-долу:



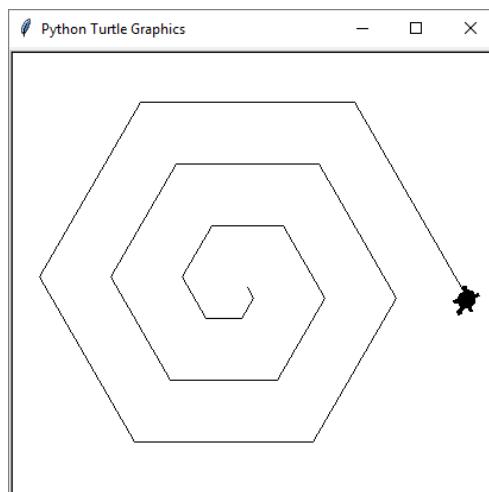
Подсказка: сменете цвета: `my_turtle.color("green")`.

В цикъл повторете 5 пъти следното:

- Движение напред 200.
- Ротация на 144 градуса.

### Задача \* чертане на спирала с костенурката

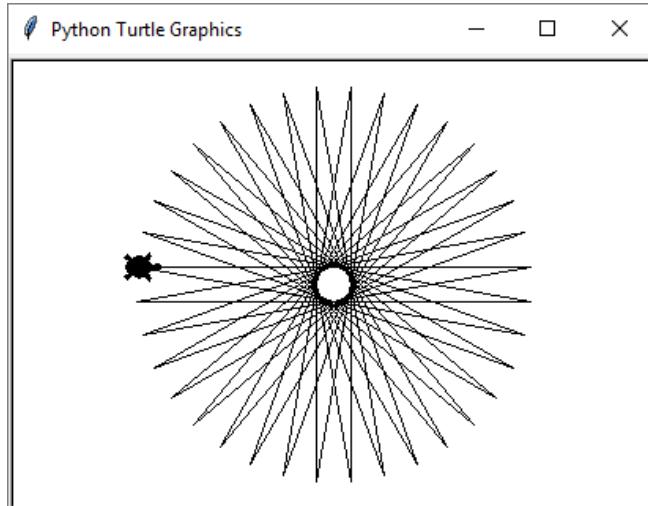
Добавете нов Python файл `spiral.py`, който чертае спирала с 20 върха като на фигурата по-долу:



**Подсказка:** Чертайте в цикъл, като движите напред и завъртате. С всяка стъпка увеличавайте постепенно дължината на движението напред и завъртайте на 60 градуса.

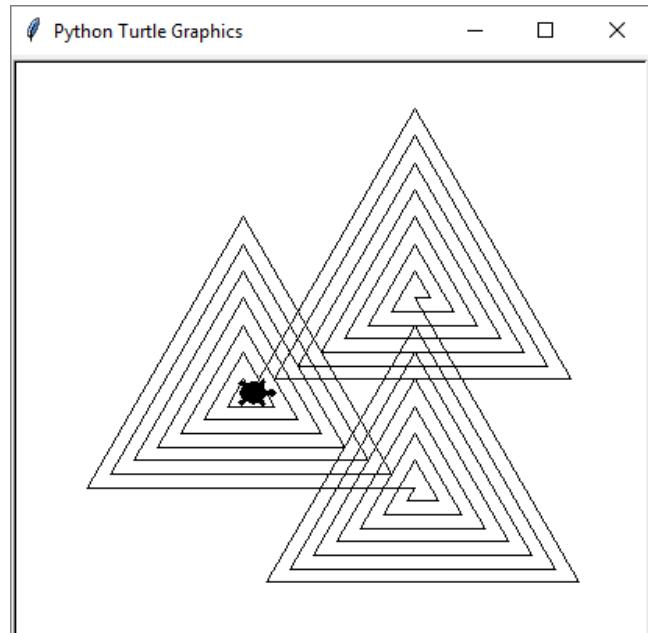
### Задача: \* чертане на слънце с костенурката

Добавете нов Python файл `sun.py`, който чертае слънце с 36 върха като на фигурата по-долу:



### Задача: \* чертане на спирален триъгълник с костенурката

Добавете нов Python файл `triangle.py`, който чертае три триъгълника с по 22 върха като на фигурата по-долу:



**Подсказка:** Чертайте в цикъл като движите напред и завъртате. С всяка стъпка увеличавайте с 10 дължината на движението напред и завъртайте на 120 градуса. Повторете 3 пъти за трите триъгълника.

Ако имате проблеми с примерния проект по-горе, питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

# Глава 5.2. Повторения (цикли) – изпитни задачи

В предходната глава научихме как да изпълним даден блок от команди **повече от веднъж**. Затова въведохме **for цикъл** и разгледахме някои от основните му приложения. Целта на настоящата глава е да затвърдим знанията си, решавайки няколко по-сложни задачи с цикли, давани на приемни изпити. За някои от тях ще покажем примерни подробни решения, а за други ще оставим само напътствия. Преди да се захванем за работа е добре да си припомним конструкцията на цикъла **for**:



**for** циклите се състоят от:

- Инициализационен блок, в който се декларира променливата-брояч (**i**) и с помощта на вградената функция **range(...)** в Python, определяме каква да бъде началната и крайната ѝ стойност.
- Обновяване на брояча – подава се като трети параметър на функцията **range(...)** и указва с каква стъпка да се обновява променливата-брояч.
- Тяло на цикъла - съдържа произволен блок със сорс код.

## Изпитни задачи

Да решим няколко задачи с цикли от изпити в СофтУни.

### Задача: хистограма

Дадени са **n** цели числа в интервала [1 ... 1000]. От тях някакъв процент **p1** са под 200, процент **p2** са от 200 до 399, процент **p3** са от 400 до 599, процент **p4** са от 600 до 799 и останалите **p5** процента са от 800 нагоре. Да се напише програма, която изчислява и отпечатва процентите **p1**, **p2**, **p3**, **p4** и **p5**.

### Входни данни

На първия ред от входа стои цялото число **n** ( $1 \leq n \leq 1000$ ), което представлява броя редове с числа, които ще ни бъдат подадени. На следващите **n** реда стои **по едно цяло число** в интервала [1 ... 1000] – числата, върху които да бъде изчислена хистограмата.

## Изходни данни

Да се отпечата на конзолата **хистограма от 5 реда**, всеки от които съдържа число между 0% и 100%, форматирано с точност две цифри след десетичния знак (например 25.00%, 66.67%, 57.14%).

**Пример:** имаме  $n = 20$  числа: 53, 7, 56, 180, 450, 920, 12, 7, 150, 250, 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение:

Група	Числа	Брой числа	Процент
< 200	53, 7, 56, 180, 12, 7, 150, 2, 199, 46, 128, 65	12	$p1 = 12 / 20 * 100 = 60.00\%$
200... 399	250, 200	2	$p2 = 2 / 20 * 100 = 10.00\%$
400... 599	450	1	$p3 = 1 / 20 * 100 = 5.00\%$
600... 799	680, 600, 799	3	$p4 = 3 / 20 * 100 = 15.00\%$
$\geq 800$	920, 800	2	$p5 = 2 / 20 * 100 = 10.00\%$

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
9	33.33%	14	57.14%	7	14.29%
367	33.33%	53	14.29%	800	28.57%
99	11.11%	7	7.14%	801	14.29%
200	11.11%	56	14.29%	250	14.29%
799	11.11%	180	7.14%	199	28.57%
999		450		399	
333		920		599	
555		12		799	
111		7			
9		150			
		250			
		680			
		2			
		600			
		200			

Вход	Изход	Вход	Изход
3	66.67%	4	75.00%
1	0.00%	53	0.00%
2	0.00%	7	0.00%
999	0.00%	56	0.00%
	33.33%	999	25.00%

## Насоки и подсказки

Програмата, която решава този проблем, можем да разделим мислено на три части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на числото  $n$ , последвано от  $n$  на брой цели числа, всяко на отделен ред.
- **Обработка на входните данни** – в случая това означава разпределение на числата по групи и изчисляване на процентното разделение по групи.
- **Извеждане на краен резултат** – отпечатване на хистограмата на конзолата в посочения формат.

### Прочитане на входните данни

Преди да преминем към самото прочитане на входните данни трябва да си **декларирате променливите**, в които ще ги съхраняваме:

```
# Променливи, в които ще запазим
# процентното разделение на отделните групи
p1_percentage = 0
p2_percentage = 0
p3_percentage = 0
p4_percentage = 0
p5_percentage = 0

# Променливи, пазещи броя числа по групи
cnt_p1 = 0
cnt_p2 = 0
cnt_p3 = 0
cnt_p4 = 0
cnt_p5 = 0
```

Декларирате си променливи **p1\_percentage**, **p2\_percentage** и т.н., в които ще пазят процентите, както и **cnt\_p1**, **cnt\_p2** и т.н., в които ще пазим броя на числата от съответната група.

След като сме си декларирали нужните променливи, можем да пристъпим към прочитането на числото **n** от конзолата:

```
n = int(input())
```

## Обработка на входните данни

За да прочетем и разпределим всяко число в съответната му група, ще си послужим с **for цикъл** от 0 до **n** (броя на числата). Всяка итерация на цикъла ще прочита и разпределя **едно единствено** число (**current\_number**) в съответната му група. За да определим дали едно число принадлежи към дадена група, **правим проверка в съответния й диапазон**. Ако това е така - увеличаваме броя на числата в тази група (**cnt\_p1**, **cnt\_p2** и т.н.) с 1:

```
for index in range(n):
    current_number = int(input())

    if current_number < 200:
        cnt_p1 += 1
    elif 200 <= current_number <= 399:
        cnt_p2 += 1
    elif 400 <= current_number <= 599:
        cnt_p3 += 1
    elif 600 <= current_number <= 799:
        cnt_p4 += 1
    else:
        cnt_p5 += 1
```

След като сме определили колко числа има във всяка група, можем да преминем към изчисляването на процентите, което е и главна цел на задачата. За това ще използваме следната формула:

$$(\text{процент на група}) = (\text{брой числа в група}) * 100 / (\text{брой на всички числа})$$

Няма значение дали ще разделим на **100** (число тип **int**) или на **100.0** (число тип **float**), **делението** ще се извърши и в променливата ще се запази резултата от него. Например: **5 / 2 = 2.5**, а **5 / 2.0 = 2.5**. В **Python 3**, няма значение дали ще се дели на цяло число или реално, ако резултата е реално число, то той ще се запише в променливата като число с плаваща запетая. Но, в **Python 2.7** е необходимо първо числата да се сведат до тип **float**, за да получим правилен резултат - реално число. Имайки това предвид, формулата за първата променлива ще изглежда така:

```
p1_percentage = cnt_p1 * 100 / n
# TODO: Добавете формулите и за останалите променливи
```

За да стане още по-ясно какво се случва, нека разгледаме примера по-долу.

В случая **n = 3**. За цикъла имаме:

- **i = 0** – прочитаме числото 1, което е по-малко от 200 и попада в първата група, следователно увеличаваме брояча на групата (**cnt\_p1**) с 1.
- **i = 1** – прочитаме числото 2, което отново попада в първата група и увеличаваме брояча ѝ (**cnt\_p1**) отново с 1.
- **i = 2** – прочитаме числото 999, което попада в последната група (**p5**), защото е по-голямо от 800, и увеличаваме брояча на групата (**cnt\_p5**) с 1.

Вход	Изход
3	66.67%
1	0.00%
2	0.00%
999	33.33%

След прочитането на числата в първата група имаме 2 числа, а в последната – имаме 1 число. В другите групи **нямаме числа**. Като приложим гореспоменатата формула, изчисляваме процентите на всяка група. Няма значение как ще умножим по **100**, или по **100.0** ще получим един и същ резултат: за **първата** група **66.67%**, а за **последната** група – **33.33%**. Но все пак да споменем отново, че това е валидно само за **Python 3**.

## Извеждане на краен резултат

Остава само да отпечатаме получените резултати. В условието е казано, че процентите трябва да са **с точност две цифри след десетичната точка**. Това ще постигнем, като след placeholder-а изпишем **.2f**:

```
print("%.2f" % p1_percentage + "%")
```



## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#0>.

## Задача: умната Лили

Лили вече е на **N** години. За всеки свой **рожден ден** тя получава подарък. За **нечетните** рождения дни (1, 3, 5, ..., n) получава **играчки**, а за всеки **четен** (2, 4, 6, ..., n) получава  **pari**. За **втория рожден ден** получава **10.00 лв.**, като **сумата се увеличава с 10.00 лв.** за всеки следващ четен рожден ден (2 -> 10, 4 -> 20, 6 -> 30 и т.н.). През годините Лили тайно е спестявала парите. **Братът** на Лили, в годините, които тя **получава пари**, взима по **1.00 лев** от тях. Лили **продала играчките**, получени през годините, **всяка за P лева** и добавила сумата към спестените пари. С парите искала да си **купи пералня за X лева**. Напишете програма, която да пресмята колко пари е събрала и дали ѝ стигат да купи пералня.

## Входни данни

От конзолата се прочитат **3 числа**, всяко на отделен ред:

- Възрастта на Лили – **цяло число** в интервала [1 ... 77].
- Цената на пералнята – **число** в интервала [1.00 ... 10 000.00].
- Единична цена на играчка – **цяло число** в интервала [0 ... 40].

## Изходни данни

Да се отпечата на конзолата един ред:

- Ако парите на Лили са достатъчни:
  - "Yes! {N}" – където **N** е остатъка пари след покупката
- Ако парите не са достатъчни:
  - "No! {M}" – където **M** е сумата, която не достига
- Числата **N** и **M** трябва да са **форматирани до втория знак след десетичната точка**.

## Примерен вход и изход

Вход	Изход	Коментари
21 1570.98 3	No! 997.98	Спестила е 550 лв.. Продала е 11 играчки по 3 лв. = 33 лв. Брат ѝ взимал 10 години по 1 лев = 10 лв. Останали $550 + 33 - 10 = 573$ лв. $573 < 1570.98$ – не е успяла да купи пералня. Не ѝ достигат $1570.98 - 573 = 997.98$ лв.

Вход	Изход	Коментари
10 170.00 6	Yes! 5.00	Първи рожден ден получава играчка; 2ти -> 10 лв.; 3ти -> играчка; 4ти -> $10 + 10 = 20$ лв.; 5ти -> играчка; 6ти -> $20 + 10 = 30$ лв.; 7ми -> играчка; 8ми -> $30 + 10 = 40$ лв.; 9ти -> играчка; 10ти -> $40 + 10 = 50$ лв. Спестила е -> $10 + 20 + 30 + 40 + 50 = 150$ лв.. Продала е 5 играчки по 6 лв. = 30 лв.. Брат ѝ взел 5 пъти по 1 лев = 5 лв. Остават -> $150 + 30 - 5 = 175$ лв. $175 \geq 170$ (цената на пералнята) успяла е да я купи и са ѝ останали $175 - 170 = 5$ лв.

## Насоки и подсказки

Решението на тази задача, подобно на предходната, също можем да разделим мислено на три части – **прочитане** на входните данни, **обработката** им и **извеждане** на резултат.

Както вече знаем, в повечето скриптови езици, както и в Python, не се грижим да определяме типа на променливите, които декларираме. Интерпретаторът сам преценява какви да бъдат те. За годините на Лили (**age**) и единичната цена на играчката (**present\_price**) по условие е дадено, че ще са **цели числа**. Затова ще използваме **вградената функция int()** да конвертираме прочетената стрингова стойност в тип цяло число. Когато се използва функция **input()**, резултатът въведен в конзолата винаги е от тип текст (**string**), затова ако е необходимо той да бъде конвертиран към друг тип, можем да използваме **вградените в Python функции** за целта. За цената на пералнята (**price\_of\_washing\_machine**) знаем, че е **дробно число и избираме да използваме тип float**. В кода по-долу декларираме и **инициализираме** (присвояваме стойност) на променливите:

```
age = int(input())
price_of_washing_machine = float(input())
present_price = int(input())
```

За да решим задачата, ще се нуждаем от няколко помощни променливи – за **броя на играчките** (**number\_of\_toys**), за **спестените пари** (**saved\_money**) и за **парите, получени на всеки рожден ден** (**money\_for\_birthday**). Като присвояваме на **money\_for\_birthday** първоначална стойност 10, тъй като по условие е дадено, че първата сума, която Лили получава, е 10 лв:

```
number_of_toys = 0
saved_money = 0
money_for_birthday = 10
```

С **for цикъл** преминаваме през всеки рожден ден на Лили. Когато водещата променлива е **четно число**, това означава, че Лили е **получила пари** и съответно прибавяме тези пари към общите ѝ спестявания. Едновременно с това **изваждаме по 1 лев** – парите, които брат ѝ взема. След това **увеличаваме** стойността на променливата **money\_for\_birthday**, т.е. увеличаваме с 10 сумата, която тя ще получи на следващия си рожден ден. Обратно, когато водещата променлива е **нечетно число**, увеличаваме броя на **играчките**. Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2 – когато остатъкът е 0, числото е четно, а при остатък 1 – нечетно:

```
for current_year in range(1, age + 1):
    if current_year % 2 == 0:
        saved_money += (money_for_birthday - 1)
        money_for_birthday += 10
    else:
        number_of_toys += 1
```

Към спестяванията на Лили прибавяме и парите от продадените играчки:

```
saved_money += number_of_toys * present_price
```

Накрая остава да отпечатаме получените резултати, като се съобразим с форматирането, указано в условието, т.е. сумата трябва да е **закръглена до две цифри след десетичния знак**:

```
print("Yes! %.2f" % (saved_money - price_of_washing_machine))
    if saved_money >= price_of_washing_machine
    else "No! %.2f" % (price_of_washing_machine - saved_money))
```

В някои програмни езици съществува конструкция, като **условния оператор** (`? :`) (наричан още тернарен оператор), тъй като записът е по-кратък. В Python синтаксисът му е следният: **операнд1 if операнд2 else операнд3**. Вторият операнд е нашето условие и трябва да е от **булев тип** (т.е. да връща `true/false`). Ако **операнд2** върне стойност `true`, то ще се изпълни **операнд1**, а ако върне `false` – **операнд3**. В нашия случай проверяваме дали **събраните пари** от Лили стигат за една пералня. Ако те са повече или равни на цената на пералнята, то проверката `saved_money >= price_of_washing_machine` ще върне `true` и ще се отпечата "Yes! ...", а ако е по-малко – резултатът ще е `false` и ще се отпечата "No! ...". Разбира се, вместо условния оператор, можем да използваме и обикновени **if** проверки.

Прочетете повече за условния оператор: от тази статия [http://book.pythontips.com/en/latest/ternary\\_operators.html](http://book.pythontips.com/en/latest/ternary_operators.html).

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#1>.

## Задача: завръщане в миналото

Иванчо е на **18 години** и получава наследство, което се състои от **X** **сума пари** и **машина** на времето. Той решава **да се върне до 1800 година**, но не знае **дали парите ще са достатъчни**, за да живее без да работи. Напишете **програма**, която пресмята дали Иванчо ще има достатъчно пари, за да не се налага да работи **до дадена година включително**. Като приемем, че за **всяка четна** (1800, 1802 и т.н.) година ще харчи **12 000** **долара**. За **всяка нечетна** (1801, 1803 и т.н.) ще харчи **12 000 + 50 \* [годините, които е навършил през дадената година]**.

## Входни данни

Входът се чете от конзолата и **съдържа точно 2 реда**:

- **Наследените пари** – реално число в интервала [1.00 ... 1 000 000.00].
- **Годината, до която трябва да живее (включително)** – цяло число в интервала [1801 ... 1900].

## Изходни данни

Да се отпечатана на конзолата 1 ред. Сумата трябва да е форматирана до два знака след десетичния знак:

- Ако парите са достатъчно:
  - "Yes! He will live a carefree life and will have {N} dollars left." – където N са парите, които ще му останат.
- Ако парите НЕ са достатъчно:
  - "He will need {M} dollars to survive." – където M е сумата, която НЕ достига.

## Примерен вход и изход

Вход	Изход	Обяснения
50000 1802	Yes! He will live a carefree life and will have 13050.00 dollars left.	1800 → четна → Харчи 12000 долара → Остават $50000 - 12000 = 38000$ 1801 → нечетна → Харчи $12000 + 19 * 50 = 12950$ → Остават $38000 - 12950 = 25050$ 1802 → четна → Харчи 12000 → Остават $25050 - 12000 = 13050$
100000.15 1808	He will need 12399.85 dollars to survive.	1800 → четна → Остават $100000.15 - 12000 = 88000.15$ 1801 → нечетна → Остават $88000.15 - 12950 = 75050.15$ ... 1808 → четна → $-399.85 - 12000 = -12399.85$ 12399.85 не достигат

## Насоки и подсказки

Методът за решаване на тази задача не е по-различен от тези на предходните, затова започваме **деклариране и инициализиране** на нужните променливи. В условието е казано, че годините на Иванчо са 18, ето защо при декларацията на променливата **years** ѝ задаваме начална стойност **18**. Другите променливи прочитаме от конзолата:

```
heritage = ...
year_to_live = ...
years = ...
```

С помощта на **for** цикъл ще обходим всички години. Започваме от 1800 – годината, в която Иванчо се връща, и стигаме до годината, до която той трябва да живее. В цикъла проверяваме дали текущата година е четна или нечетна. Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2. Ако годината е четна, изваждаме от наследството (**heritage**) 12000, а ако е нечетна, изваждаме от наследството (**heritage**)  $12000 + 50 * (\text{годините на Иванчо})$ :

```
for current_year in range(1800, year_to_live + 1):
    if current_year % 2 == 0:
        heritage -= 12000
    else:
        heritage -= (12000 + 50 * years)
    years += 1
```

Накрая остава да отпечатаме резултатите, като за целта правим проверка дали наследството (**heritage**) му е било достатъчно да живее без да работи или не. Ако наследството (**heritage**) е положително число, отпечатваме: "**Yes! He will live a carefree life and will have {N} dollars left.**", а ако е отрицателно число: "**He will need {M} dollars to survive.**". Не забравяме да форматираме сумата до два знака след десетичната точка.

**Hint:** Обмислете използването на функцията **abs(...)** при отпечатване на изхода, когато наследството е недостатъчно.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#2>.

## Задача: болница

За даден период от време, всеки ден в болницата пристигат пациенти за преглед. Тя разполага **първоначално** със **7 лекари**. Всеки лекар може да преглежда **само по един пациент на ден**, но понякога има недостиг на лекари, затова **останалите пациенти се изпращат в други болници**. Всеки **трети ден** болницата прави **изчисления** и ако броят на непрегледаните пациенти е **по-голям** от броя на прегледаните, се назначава **още един лекар**. Като назначаването става преди да започне приемът на пациенти за деня.

Напишете програма, която изчислява **за дадения период** броя на прегледаните и непрегледаните пациенти.

## Входни данни

Входът се чете от **конзолата** и съдържа:

- На първия ред – **периода**, за който трябва да направите изчисления. **Цяло число** в интервала [1 ... 1000].
- На следващите редове (равни на броя на дните) – **броя пациенти**, които пристигат за **текущия ден**. Цяло число в интервала [0 ... 10 000].

## Изходни данни

Да се отпечатат на конзолата 2 реда:

- На първия ред: "Treated patients: {брой прегледани пациенти}."
- На втория ред: "Untreated patients: {брой непрегледани пациенти}."

## Примерен вход и изход

Вход	Изход	Вход	Изход
6	Treated patients: 40. Untreated patients: 87.	3	Treated patients: 21. Untreated patients: 0.
25		7	
25		7	
25		7	
25			
25			
2			

Вход	Изход	Обяснения
4	Treated patients: 23.	1 ден: 7 прегледани и 0 непрегледани пациент за деня
7	Untreated patients: 21.	2 ден: 7 прегледани и 20 непрегледани пациент за деня
27		3 ден: До момента прегледаните пациенти са общо 14, а непрегледаните – 20 → Назначава се нов лекар → 8 прегледани и 1 непрегледан пациент за деня
9		4 ден: 1 прегледан и 0 непрегледани пациент за деня
1		Общо: 23 прегледани и 21 непрегледани пациенти.

## Насоки и подсказки

Отново започваме, като **декларираме и инициализираме** нужните променливи. Периодът, за който трябва да направим изчисленията, прочитаме от конзолата и

запазваме в променливата **period**. Ще се нуждаем и от няколко помощни променливи: броя на излекуваните пациенти (**treated\_patients**), броя на неизлекуваните пациенти (**untreated\_patients**) и броя на докторите (**count\_of\_doctors**), който първоначално е 7:

```
period = 7

treated_patients = 0
untreated_patients = 0
count_of_doctors = 7
```

С помощта на **for** цикъл обхождаме всички дни в дадения период (**period**). За всеки ден прочитаме от конзолата броя на пациентите (**current\_patients**). Увеличаването на докторите по условие може да стане **всеки трети ден**, но само ако броят на непрегледаните пациенти е **по-голям** от броя на прегледаните. За тази цел проверяваме дали денят е трети – чрез аритметичния оператор за деление с остатък (%): **day % 3 == 0**.

Например:

- Ако денят е **трети**, остатъкът от делението на 3 ще бъде 0 (**3 % 3 == 0**) и проверката **day % 3 == 0** ще върне **true**.
- Ако денят е **втори**, остатъкът от делението на 3 ще бъде 2 (**2 % 3 == 2**) и проверката ще върне **false**.
- Ако денят е **четвърти**, остатъкът от делението ще бъде 1 (**4 % 3 == 1**) и проверката отново ще върне **false**.

Ако проверката **day % 3 == 0** върне **true**, ще се провери дали и броят на неизлекуваните пациенти е по-голям от броя на излекуваните пациенти: **untreated\_patients > treated\_patients**. Ако резултатът отново е **true**, тогава ще се увеличи броят на лекарите (**count\_of\_doctors**).

След това проверяваме броя на пациентите за деня (**current\_patients**) дали е по-голям от броя на докторите (**count\_of\_doctors**). Ако броят на пациентите е **по-голям**:

- Увеличаваме стойността на променливата **treated\_patients** с броя на докторите (**count\_of\_doctors**).
- Увеличаваме стойността на променливата **untreated\_patients** с броя на останалите пациенти, който изчисляваме, като от всички пациенти извадим броя на докторите (**current\_patients - count\_of\_doctors**).

Ако броят на пациентите **не е по-голям**, увеличаваме само променливата **treated\_patients** с броя на пациентите за деня (**current\_patients**):

```
for day in range(1, period + 1):
    current_patients = ...
```

```

if (day % 3 == 0) and (untreated_patients > treated_patients):
    count_of_doctors += 1

if current_patients > count_of_doctors:
    treated_patients += count_of_doctors
    untreated_patients += current_patients - count_of_doctors
else:
    treated_patients += current_patients

```

Накрая трябва само да отпечатаме броя на излекуваните и броя на неизлекуваните пациенти.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#3>.

## Задача: деление без остатък

Дадени са  $n$  цели числа в интервала  $[1 \dots 1000]$ . От тях някакъв процент  $p1$  се делят без остатък на 2, процент  $p2$  се делят без остатък на 3, процент  $p3$  се делят без остатък на 4. Да се напише програма, която изчислява и отпечатва процентите  $p1$ ,  $p2$  и  $p3$ . Пример: имаме  $n = 10$  числа: 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение и визуализация:

Деление без остатък на:	Числа	Брой	Процент
2	680, 2, 600, 200, 800, 46, 128	7	$p1 = (7 / 10) * 100 = 70.00\%$
3	600	1	$p2 = (1 / 10) * 100 = 10.00\%$
4	680, 600, 200, 800, 128	5	$p3 = (5 / 10) * 100 = 50.00\%$

## Входни данни

На първия ред от входа стои цялото число  $n$  ( $1 \leq n \leq 1000$ ) – брой числа. На следващите  $n$  реда стои по едно цяло число в интервала  $[1 \dots 1000]$  – числата, които да бъдат проверени на колко се делят.

## Изходни данни

Да се отпечатат 3 реда, всеки от които съдържа процент между 0% и 100%, с точност две цифри след десетичния знак, например 25.00%, 66.67%, 57.14%.

- На първия ред – процентът на числата, които се делят на 2.
- На втория ред – процентът на числата, които се делят на 3.
- На третия ред – процентът на числата, които се делят на 4.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
10	70.00%	3	33.33%	1	100.00%
680	10.00%	3	100.00%	12	100.00%
2	50.00%	6	0.00%		100.00%
600		9			
200					
800					
799					
199					
46					
128					
65					

## Насоки и подсказки

За тази и следващата задача ще трябва сами да напишете програмния код, следвайки дадените напътствия.

Програмата, която решава текущия проблем, е аналогична на тази от задача **Хистограма**, която разгледахме по-горе. Затова можем да започнем с декларацията на нужните ни променливи. Примерни имена на променливи може да са: **n** – брой на числата (който трябва да прочетем от конзолата) и **divisible\_by\_2**, **divisible\_by\_3**, **divisible\_by\_4** – помощни променливи, пазещи броя на числата от съответната група.

За да прочетем и разпределим всяко число в съответната му група, ще трябва да завъртим **for цикъл** от **0** до **n** (броя на числата). Всяка итерация на цикъла трябва да прочита и разпределя **едно единствено число**. Различното тук е, че **едно число може да попадне в няколко групи едновременно**, затова трябва да направим **три отделни if проверки за всяко число** – съответно дали се дели на 2, 3 и 4 и да увеличим стойността на променливата, която пази броя на числата в съответната група.

**Внимание:** **if-elif** конструкция в този случай няма да ни свърши работа, защото след като намери съвпадение се прекъсва по-нататъшното проверяване на условията.

Накрая трябва да отпечатате получените резултати, като спазвате посочения формат в условието.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#4>.

## Задача: логистика

Отговаряте за логистиката на различни товари. В зависимост от теглото на всеки товар е нужно различно превозно средство и струва различна цена на тон:

- До 3 тона – **микробус** (200 лева на тон).
- От над 3 и до 11 тона – **камион** (175 лева на тон).
- Над 11 тона – **влак** (120 лева на тон).

Вашата задача е да изчислите **средната цена на тон превозен товар**, както и колко процента от товара се превозват с **всяко превозно средство**.

### Входни данни

От конзолата се четат **поредица от числа**, всяко на отделен ред:

- Първи ред: брой на товарите за превоз – **цяло число** в интервала [1 ... 1000].
- На всеки следващ ред се подава **тонажът** на **последния товар** – **цяло число** в интервала [1 ... 1000].

### Изходни данни

Да се отпечатат на конзолата **4 реда**, както следва:

- Ред #1 – средната цена на тон превозен товар (закръглена до втория знак след десетичната точка).
- Ред #2 – процентът товар, превозан с **микробус** (между 0.00% и 100.00%, закръглен до втория знак след десетичната точка).
- Ред #3 – процентът товар, превозвани с **камион** (между 0.00% и 100.00%).
- Ред #4 – процентът товар, превозвани с **влак** (между 0.00% и 100.00%).

### Примерен вход и изход

Вход	Изход	Обяснения
4	143.80	С <b>микробус</b> се превозват два от товарите <b>1 + 3</b> , общо <b>4</b> тона.
1	16.00%	С <b>камион</b> се превозва един от товарите: <b>5</b> тона.
5	20.00%	С <b>влак</b> се превозва един от товарите: <b>16</b> тона.
16	64.00%	Сумата от всички товари е: $1 + 5 + 16 + 3 = 25$ тона. Процент товар с <b>микробус</b> : $4/25 * 100 = 16.00\%$ Процент товар с <b>камион</b> : $5/25 * 100 = 20.00\%$ Процент товар с <b>влак</b> : $16/25 * 100 = 64.00\%$
3		Средна цена на тон превозен товар: $(4 * 200 + 5 * 175 + 16 * 120) / 25 = 143.80$

## Насоки и подсказки

Първо ще прочетем теглото на всеки товар и ще сумираме колко тона се превозват съответно с микробус, камион и влак и ще изчислим и общите тонове превозени товари. Ще пресметнем цените за всеки вид транспорт според превозените тонове и общата цена. Накрая ще пресметнем и отпечатаме общата средна цена на тон и каква част от товара е превозена с всеки вид транспорт процентно.

Декларираме си нужните променливи, например: `count_of_loads` – броя на товарите за превоз (прочитаме ги от конзолата), `sum_of_tons` – сумата от тонажа на всички товари, `microbus_tons`, `truck_tons`, `train_tons` – променливи, пазещи сумата от тонажа на товарите, превозвани съответно с микробус, камион и влак.

Ще ни трябва `for` цикъл от `0` до `count_of_loads - 1`, за да обходим всички товари. За всеки товар прочитаме теглото му (в тонове) от конзолата и го запазваме в променлива, например `tons`. Прибавяме към сумата от тонажа на всички товари (`sum_of_tons`) теглото на текущия товар (`tons`). След като сме прочели теглото на текущия товар, трябва да определим кое превозно средство ще се ползва за него (микробус, камион или влак). За целта ще ни трябват `if-elif` проверки:

- Ако стойността на променливата `tons` е по-малка от `3`, увеличаваме стойността на променливата `microbus_tons` със стойността на `tons`:  
`microbus_tons += tons;`
- Иначе, ако стойността `tons` е до `11`, увеличаваме `truck_tons` с `tons`.
- Ако `tons` е повече от `11`, увеличаваме `train_tons` с `tons`.

Преди да отпечатаме изхода трябва да изчислим процента на тоновете, превозвани с всяко превозно средство и средната цена на тон. За средната цена на тон ще си декларираме още една помощна променлива `total_price`, в която ще сумираме общата цена на всички превозвани товари (с микробус, камион и влак). Средната цена ще получим, разделяйки `total_price` на `sum_of_tons`. Остава сами да изчислите процента на тоновете, превозвани с всяко превозно средство, и да отпечатате резултатите, спазвайки формата в условието.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1054#5>.

# Глава 6.1. Вложени цикли

В настоящата глава ще разгледаме **вложените цикли** и как да използваме **for** цикли за **чертане** на различни **фигурки на конзолата**, които се състоят от символи и знаци, разположени в редове и колони на конзолата. Ще използваме **единични** и **вложени цикли** (цикли един в друг), **изчисления и проверки**, за да отпечатваме на конзолата прости и не чак толкова прости фигурки по зададени размери.

## Видео

Гледайте видео-урок по тази глава тук: <https://youtube.com/watch?v=kkxl1bitNAg>.

### Пример: правоъгълник от $10 \times 10$ звездички

Да се начертате в конзолата правоъгълник от  $10 \times 10$  звездички.

Вход	Изход
(няма)	***** ***** ***** ***** ***** ***** ***** ***** ***** *****

#### Насоки и подсказки

```
for i in range(10):  
    print('*' * 10)
```

Как работи примерът? Инициализира се **цикъл с променлива *i***. Началната стойност по подразбиране на променливата е ***i = 0***. С всяка итерация на цикъла променливата се увеличава с **1**, докато е **по-малка от 10**. Така кодът в тялото на цикъла се изпълнява **10 пъти** - от **0-ия** до **9-ия** път включително. В тялото на цикъла се печата на нов ред в конзолата **'\*' \* 10**, което създава низ (стринг) от 10 звездички.

#### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#0>.

### Пример: правоъгълник от $N \times N$ звездички

Да се напише програма, която въвежда цяло положително число **n** и печата на конзолата **правоъгълник от  $N \times N$  звездички**.

Вход	Изход	Вход	Изход	Вход	Изход
2	** **	3	*** *** ***	4	**** **** **** ****

## Насоки и подсказки

Задачата е аналогична с предходната и решението е почти същото:

```
n = int(input())
for i in range(n):
    print('*' * n)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#1>.

## Вложени цикли

Вложените цикли представляват конструкция, при която **в тялото на един цикъл** (външен) **се изпълнява друг цикъл** (вътрешен). За всяко завъртане на външния цикъл, вътрешният се извърта **целият**. Това се случва по следния начин:

- При стартиране на изпълнение на вложени цикли първо **стартира външният цикъл**: извършва се **инициализация** на неговата управляваща променлива и след проверка за край на цикъла, се изпълнява кодът в тялото му.
- След това се **изпълнява вътрешният цикъл**. Извършва се инициализация на началната стойност на управляващата му променлива, прави се проверка за край на цикъла и се изпълнява кодът в тялото му.
- При достигане на зададената стойност за **край на вътрешния цикъл**, програмата се връща една стъпка нагоре и се продължава започналото изпълнение на външния цикъл. Променя се с една стъпка управляващата променлива за външния цикъл, проверява се дали условието за край е удовлетворено и **започва ново изпълнение на вложения (вътрешния) цикъл**.
- Това се повтаря, докато променливата на външния цикъл достигне условието за **край на цикъла**.

Ето и един пример, с който нагледно да илюстрираме вложените цикли:

```
n = int(input())
for row in range(1, n + 1):
    for col in range(1, n + 1):
        print("*", end="")
    print()
```

Целта на кода по-горе е да се отпечата отново правоъгълник от  $N \times N$  звездички, като за всеки ред се извърта цикъл от 1 до  $N$ , а за всяка колона се извърта вложен цикъл от 1 до  $N$ .

В езика Python, когато стандартната начална стойност на променливата в цикъла (`i = 0`) не ни върши работа, можем да я променим със горепосочения синтаксис. Т.е. когато искаме цикълът да започва от 1 и да се върти до `n` включително, пишем: `for i in range(1, n + 1)`. Първата стойност в скобите указва началото на цикъла, а втората - края на цикъла, но не включително, т.е. цикълът свършва, преди да се достигне до нея.

Да разгледаме примера по-горе. След инициализацията на **първия (външен) цикъл**, започва да се изпълнява неговото **тяло**, което съдържа **втория (вложен) цикъл**. Той сам по себе си печата на един ред `n` на брой звездички. След като **вътрешният цикъл приключи** изпълнението си при първата итерация на външния, то след това **външният ще продължи**, т.е. ще отпечата един празен ред на конзолата. **След това** ще се извърши **обновяване** на променливата на **първия цикъл** и отново ще бъде изпълнен целият **втори (вложен) цикъл**. Вътрешният цикъл ще се изпълни толкова пъти, колкото се изпълнява тялото на външния цикъл, в случая `n` пъти.

## Пример: квадрат от звездички

Да се начертате на конзолата квадрат от  $N \times N$  звездички:

Вход	Изход	Вход	Изход	Вход	Изход
2	* *	3	* * * * * * * * *	4	* * * * * * * * * * * * * * * *

### Насоки и подсказки

Задачата е аналогична на предходната. Разликата тук е, че трябва да обмислим как да печатаме интервал след звездичките по такъв начин, че да няма излишни интервали в началото или края:

```
n = int(input())
for row in range(1, n + 1):
    print('* ', end=' ')
    for col in range(1, n):
        print(' *', end=' ')
    print()
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#2>.

## Пример: триъгълник от долари

Да се напише програма, която въвежда число **n** и печата **триъгълник от долари**.

Вход	Изход	Вход	Изход
3	\$ \$ \$ \$ \$ \$	4	\$ \$ \$ \$ \$ \$ \$ \$ \$ \$

### Насоки и подсказки

Задачата е сходна с тези за рисуване на **правоъгълник** и **квадрат**. Отново ще използваме **вложени цикли**, но тук има **уловка**. Разликата е в това, че **броя на колонките**, които трябва да разпечатаме, зависят от **реда**, на който се намираме, а не от входното число **n**. От примерните входни и изходни данни забелязваме, че **броят на долларите зависи** от това на кой **ред** се намираме към момента на печатането, т.е. 1 доллар означава първи ред, 3 долара означават трети ред и т.н. Нека разгледаме долния пример по-подробно. Виждаме, че **променливата** на **вложения цикъл** е обвързана с променливата на **външния**. По този начин нашата програма печата желания триъгълник:

```
n = int(input())
for row in range(n):
    print('$', end=' ')
    for col in range(row):
        print(' ', end=' ')
    print()
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#3>.

## Пример: квадратна рамка

Да се напише програма, която въвежда цяло положително число **n** и чертае на конзолата **квадратна рамка** с размер **N x N**.

Вход	Изход	Вход	Изход	Вход	Изход
4	+ - - +   - -     - -   + - - +	5	+ - - - +   - - -     - - -     - - -   + - - - +	6	+ - - - - +   - - - -     - - - -     - - - -     - - - -   + - - - - +

## Насоки и подсказки

Можем да решим задачата по следния начин:

- Четем от конзолата числото **n**.
- Отпечатваме **горната част**: първо знак **+**, после **n-2** пъти **-** и накрая знак **+**.
- Отпечатваме **средната част**: печатаме **n-2** реда като първо печатаме знак **|**, после **n-2** пъти **-** и накрая отново знак **|**. Това можем да го постигнем с вложени цикли.
- Отпечатваме **долната част**: първо **+**, после **n-2** пъти **-** и накрая **+**.

Ето и примерна имплементация на описаната идея с вложени цикли:

```
n = int(input())

# printing the top row: + - - - +
print('+', end=' ')
for i in range(n - 2):
    print(' -', end=' ')
print(' +')

# printing the mid rows: | - - - |
for row in range(n - 2):
    print('|', end=' ')
    for col in range(n - 2):
        print(' -', end=' ')
    print(' |')

# printing the bottom row: + - - - +
print('+', end=' ')
for i in range(n - 2):
    print(' -', end=' ')
print(' +')
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#4>.

## Пример: ромбче от звездички

Да се напише програма, която въвежда цяло положително число **n** и печата ромбче от звездички с размер **N**.

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	*	2	* * *	3	* * * * * * * * * * *	4	* * * * * * * * * * * * * * * *

## Насоки и подсказки

За решението на тази задача е нужно да **разделим** мислено ромба на **две части** - **горна**, която включва **и** средния ред, и **долна**. За **разпечатването** на всяка една част ще използваме **два** отделни цикъла, като оставяме на читателя сам да намери зависимостта между **n** и променливите на циклите. За първия цикъл може да използваме следните насоки:

- Отпечатваме **n-row** интервала.
- Отпечатваме **\***.
- Отпечатваме **row-1** пъти **\***.

Втората (долна) част ще разпечатаме по **аналогичен** начин, което отново оставяме на читателя да се опита да направи сам.

```
n = int(input())
```

```
for row in range(1, n + 1):
    for col in range(1, n - row + 1):
        print(' ', end='')
    print('* ', end='')

    for col in range(1, row):
        print(' *', end='')
    print()
```

```
# TODO: print the bottom side of the rhombus
```



В езика Python стандартната стъпка на **for** цикъла е положителна и е равна на 1. Ако искаме да я променим, трябва при аргументите на цикъла да използваме трети параметър: **for i in range (0, 100, 2)**. Третият параметър в случая показва, че променливата ще се увеличава от 0 до 99 включително, със стъпка 2.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#5>.

## Пример: коледна елха

Да се напише програма, която въвежда число  $n$  ( $1 \leq n \leq 100$ ) и печата коледна елха с височина  $N + 1$ .

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	*   *	2	*   * **   **	3	*   * **   ** ***   ***	4	*   * **   ** ***   *** ****   ****

### Насоки и подсказки

От примерите виждаме, че елхата може да бъде **разделена** на **три** логически части. Първата част са **звездичките и празните места преди и след тях**, средната част е **|**, а последната част са отново **звездички**, като този път **празни места има само преди тях**. Разпечатването може да бъде постигнато само с **един цикъл** и операцията **умножаване на стринг**, която ще използваме един път за звездинчките и един път за интервалите:

```
n = int(input())

for i in range(n + 1):
    stars = '*' * i
    spaces = ' ' * (n - i)
    print(spaces, end='')
    print(stars, end='')
    print(' | ', end='')
    print(stars, end='')
    print(spaces)
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#6>.

## Чертане на по-сложни фигури

Да разгледаме как можем да чертаем на конзолата фигури с по-сложна логика на конструиране, за които трябва повече да помислим, преди да почнем да пишем.

## Пример: слънчеви очила

Да се напише програма, която въвежда цяло число  $n$  ( $3 \leq n \leq 100$ ) и печата слънчеви очила с размер  $5*N \times N$  като в примерите:

Вход	Изход
3	***** * ***** *////*     *////* ***** * *****

Вход	Изход
4	***** * ***** *////*     *////* *////*     *////* ***** * *****

Вход	Изход
5	***** * ***** *////*     *////* *////*     *////*     *////*     *////* *////*     *////*     *////* ***** * *****

## Насоки и подсказки

От примерите виждаме, че очилата могат да се разделят на **три части** – горна, средна и долна. По-долу е част от кода, с който задачата може да се реши.

При рисуването на горния и долния ред трябва да се изпечатат **2 \* n** звездички, **n** интервала и **2 \* n** звездички:

```

n = int(input())

# printing the top part
print('*' * (2 * n), end=' ')
print(' ' * n, end=' ')
print('*' * (2 * n))

for i in range(n - 2):
    # TODO: print the middle part

# printing the bottom part
print('*' * (2 * n), end=' ')
print(' ' * n, end=' ')
print('*' * (2 * n))

```

При печатането на **средната** част трябва да проверим дали редът е **(n -1) // 2 - 1**, тъй като от примерите е видно, че на **този ред** трябва да печатаме **вертикални чертички** вместо интервали:

```

for i in range(n - 2):
    # TODO: print */////////*

    if i == (n - 1) // 2 - 1:
        print(' ' * n, end=' ')
    else:
        print(' ' * n, end=' ')

    # TODO: print */////////*
print()

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#7>.

## Пример: къщичка

Да се напише програма, която въвежда число  $n$  ( $2 \leq n \leq 100$ ) и печата **къщичка** с размери  $N \times N$ , точно като в примерите:

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	** 	3	-*- ***  *	4	- ** - *****   **     **	5	--*-- - *** - *****   ***     ***

## Насоки и подсказки

Разбираме от условието на задачата, че къщата е с размер  $n \times n$ . Това, което виждаме от примерните вход и изход, е, че:

- Къщичката е разделена на 2 части: **покрив** и **основа**.



- Когато  $n$  е четно число, върхът на къщичката е тъп.
- Когато  $n$  е нечетно число, **покривът** е с остър връх и с един ред по-голям от **основата**.

## Покрив

- Съставен е от звезди и тирета.
- В най-високата си част има една или две звезди, спрямо това дали **n** е четно или нечетно, както и тирета.
- В най-ниската си част има много звезди и малко или никакви тирета.
- С всеки един ред по-надолу звездите се увеличават с 2, а тиретата намаляват също с 2.

## Основа

- Дълга е **n** на брой реда.
- Съставена е от звезди и вертикални черти.
- Редовете са съставени от 2 вертикални черти - по една в началото и в края на реда, както и звезди между чертите с дължина на низа **n - 2**.

Прочитаме входното число **n** от конзолата и записваме стойността му в променлива:

```
n = int(input())
```



Много е важно да проверяваме дали са валидни входните данни! В тези задачи не е проблем директно да обръщаме прочетеният вход от конзолата в тип **int**, защото изрично е казано, че ще получаваме валидни целичислени числа. Ако обаче правим по-серииозни приложения, е добра практика да проверяваме данните. Какво ще стане, ако вместо число потребителят въведе например буквата "A"?

За да начертаем покрива, записваме колко ще е началният брой звезди в променлива **stars**:

- Ако **n** е нечетно число, ще е 1 брой.
- Ако **n** е четно, ще са 2 броя.

```
stars = 1
if n % 2 == 0:
    stars += 1
```

Изчисляваме дължината на покрива. Тя е равна на половината от **n**. Резултата записваме в променливата **roof\_length**:

```
roof_length = math.ceil(n / 2)
```

Забележка: за да използваме **math.ceil()**, която закръгля към по-голямото цяло число, без значение от дробната част, е необходимо да въведем библиотеката **math**. Това става с команда **import math**. Препоръчително е да напишем **import math** (както и всички други import-и) още в началото на файла.

Важно е да съобразим, че когато **n** е нечетно число, дължината на покрива е по-голяма с един ред от тази на **основата**.

В езика **Python**, когато два целочислени типа се делят и има остатък, то резултатът ще е число с остатък. Ако искаме да извършим чисто целочислено деление без остатък, е необходимо да използваме оператора **//**.

Пример:

```
result1 = 3 / 2    # резултат 1.5
result2 = 3 // 2    # резултат 1
```

Ако искаме да закръглим резултата нагоре, трябва да използваме метода **math.ceil(...)**: **result = math.ceil(3 / 2)**.

След като сме изчислили дължината на покрива, завъртаме цикъл от 0 до **root\_length**. На всяко повторение ще:

- Изчисляваме броя **тирета**, които трябва да изрисуваме. Броят ще е равен на **(n - stars) / 2**. Записваме го в променлива **padding**.

```
padding = (n - stars) // 2
```

- Отпечатваме на конзолата: "тирета" (**padding / 2** на брой пъти) + "звезди" (**stars** пъти) + "тирета" (**padding / 2** пъти).

```
padding = (n - stars) // 2
```

```
line = '-' * padding \
      + '*' * stars \
      + '-' * padding
```

```
print(line)
```

- Преди да завърши въртенето на цикъла увеличаваме **stars** (броя на звездите) с 2.

```
stars += 2
```

След като сме приключили с покрива, е време за **основата**. Тя е по-лесна за печатане:

- Започваме с цикъл от 0 до **n** (изключено).
- Отпечатваме на конзолата: | + \* (**n - 2** на брой пъти) + |.

```
for i in range(n // 2):
    line = '|' + '*' * (n - 2) + '|'
    print(line)
```

Ако всичко сме написали както трябва, задачата ни е решена.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#8>.

## Пример: диамант

Да се напише програма, която приема цяло число  $n$  ( $1 \leq n \leq 100$ ) и печата диамант с размер  $N$ , като в следните примери:

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	*	2	**	3	- * - * - * - * -	4	- ** - * - - * - * * -	5	-- * -- - * - * - * - - - * - * - * - -- * - -

### Насоки и подсказки

Това, което знаем от условието на задачата, е че диамантът е с размер  $n \times n$ . От примерните вход и изход можем да си направим извода, че всички редове съдържат точно по  $n$  символа и всички редове, с изключение на горните върхове, имат по **2 звезди**. Можем мислено да разделим диаманта на 2 части:

- **Горна част.** Тя започва от горния връх до средата.
- **Долна част.** Тя започва от реда след средата до най-долния връх включително.

### Горна част

- Ако  $n$  е **нечетно**, то тя започва с **1 звезда**.
- Ако  $n$  е **четно**, то тя започва с **2 звезди**.
- С всеки ред надолу звездите се отдалечават една от друга.
- Пространството между, преди и след **звездите** е запълнено с **тирета**.

### Долна част

- С всеки ред надолу звездите се приближават една към друга. Това означава, че пространството (**тиретата**) между тях намалява, а пространството (**тиретата**) отляво и отдясно се увеличава.
- В най-долната си част е с **1 или 2 звезди**, спрямо това дали  $n$  е четно или не.

### Горна и долна част на диаманта

- На всеки ред звездите са заобиколени от **външни тирета**, с изключение на **средния ред**.
- На всеки ред има пространство между двете **звезди**, с изключение на първия и последния ред (понякога **звездата е 1**).

Прочитаме стойността на  $n$  от конзолата, като я конвертираме до тип **int**:

```
n = int(input())
```

Започваме да печатаме на конзолата горната част на диаманта. Първото нещо, което трябва да направим, е да изчислим началната стойност на външната бройка тирета **left\_right** (тиретата от външната част на звездите). Тя е равна на  $(n - 1) / 2$ , закръглено надолу:

```
left_right = (n - 1) // 2
```

След като сме изчислили **left\_right**, започваме да чертаем горната част на диаманта. Може да започнем, като завъртим цикъл от **0** до  $(n + 1) // 2$  (т.е. закръглено надолу).

При всяко завъртане на цикъла трябва да се изпълнят следните стъпки:

- Печатане на левите тирета на конзолата (с дължина **left\_right**) и веднага след тях първата звезда.

```
print('-' * left_right, end=' ')
print('*', end='')
```

- Изчисляване на разстоянието между двете звезди. Може да го направим като извадим от **n** дължината на външните тирета, както и числото 2 (бройката на звездите, т.е. очертанията на диаманта). Резултата от тази разлика записваме в променлива **mid**.

```
mid = n - 2 * left_right - 2
```

- Ако стойността на **mid** е по-малка от 0, то тогава знаем, че на този ред трябва да има 1 звезда. Ако е по-голяма или равно на 0, то тогава трябва да начертаем тирета с дължина **mid** и една звезда след тях.

- Печатаме на конзолата десните външни тирета с дължина също **left\_right**.

```
print('-' * left_right)
```

- В края на цикъла намаляваме стойността на **left\_right** с 1 (звездите се отдалечават).

Готови сме с горната част.

Чертането на долната част е доста подобно на това на горната част. Разликата е, че вместо да намаляваме стойността на **left\_right** с 1 към края на цикъла, ще я увеличаваме с 1 в началото на цикъла. Също така цикълът ще се върти от **0** до  $(n - 1) // 2$ :

```
for i in range((n - 1) // 2):
    left_right += 1

    print('-' * left_right, end=' ')
    print('*', end='')

mid = n - 2 * left_right - 2
```

```

if mid >= 0:
    print('-' * mid, end=' ')
    print('*', end=' ')

print('-' * left_right)

```



Повторението на код се смята за лоша практика, защото кодът става доста труден за поддръжка. Нека си представим, че имаме парче код (напр. логиката за чертането на ред от диаманта) на още няколко места и решаваме да направим промяна. За целта би било необходимо да минем през всичките места и да направим промените. А сега нека си представим, че трябва да използвате код не 1, 2 или 3 пъти, а десетки пъти. Начин за справяне с този проблем е като се използват **функции**. Можете да потърсите допълнителна информация за тях в Интернет или да прегледате [Глава 10. Функции](#).

Ако сме написали всичко коректно, задачата ни е решена.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1055#9>.

## Какво научихме от тази глава?

Запознахме се с един от начините за създаване на низове:

```
print_me = '*' * 5
```

Научихме се да чертаем фигури с вложени **for** цикли:

```

for row in range(5):
    print('*', end=' ')

    for col in range(4):
        print(' *', end=' ')

    print()

```

## Упражнения: чертане на фигурки в уеб среда

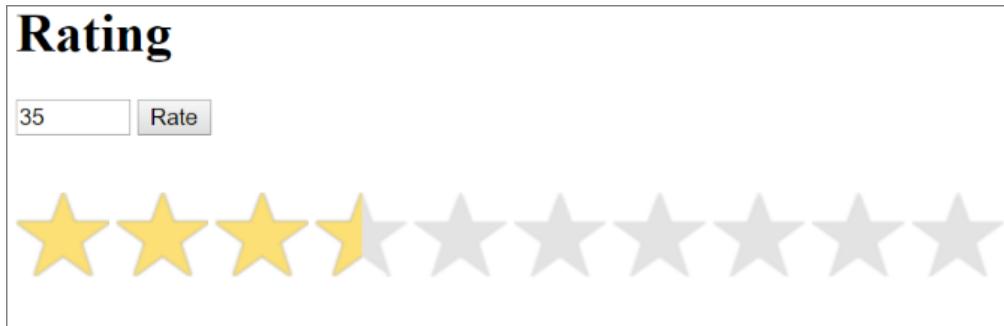
Сега, след като свикнахме с **вложените цикли** и как да ги използваме, за да чертаем фигурки на конзолата, можем да се захванем с нещо още по-интересно: да видим как циклите могат да се използват за **чертане в уеб среда**. Ще направим уеб приложение, което визуализира числовой рейтинг (число от 0 до 100) със

звездички. Такава визуализация се среща често в сайтове за електронна търговия, ревюта на продукти, оценки на събития, рейтинг на приложения и други.

Не се притеснявайте, ако не разберете целия код, как точно е направен и как точно работи проектът. Нормално е, сега се учит да пишем код, не сме стигнали до технологиите за уеб разработка. Ако имате трудности да си напишете проекта, следвайки описаните стъпки, питайте в СофтУни форума: <https://softuni.bg/forum>.

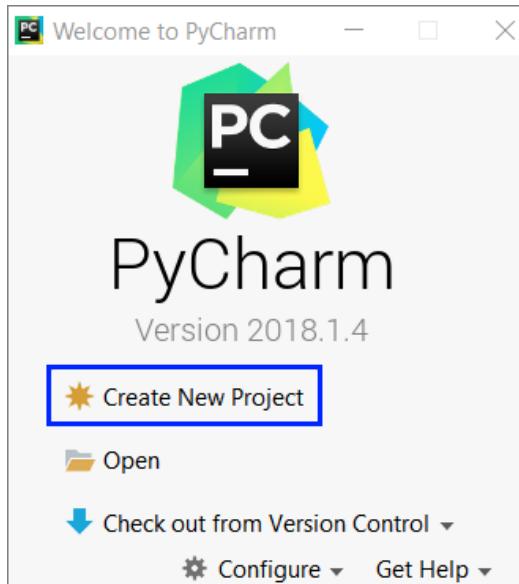
## Задача: рейтинги – визуализация в уеб среда

Да се разработи уеб приложение за визуализация на рейтинг (число от 0 до 100). Чертаят се от 1 до 10 звездички (с половинки). Звездичките да се генерираят с **for** цикъл.

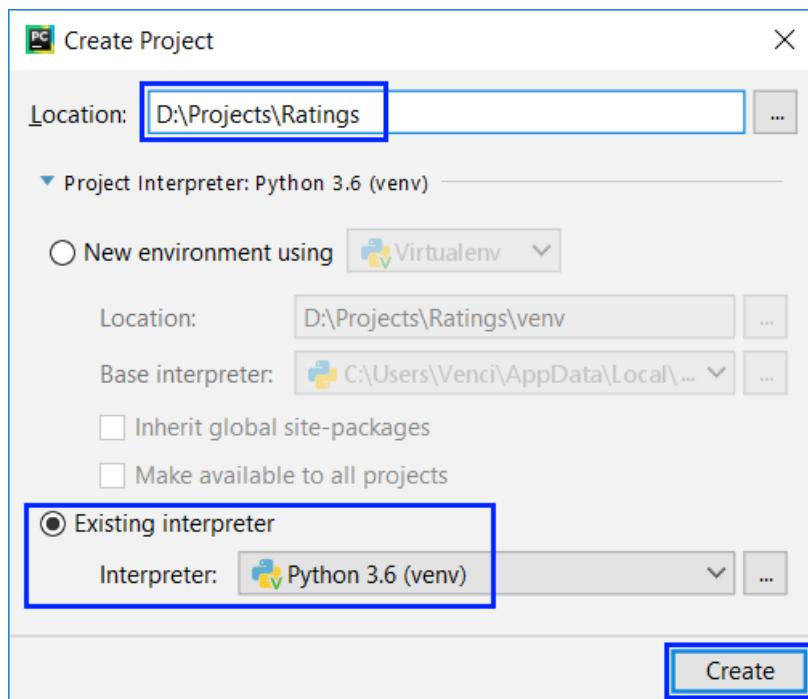


### Насоки и подсказки

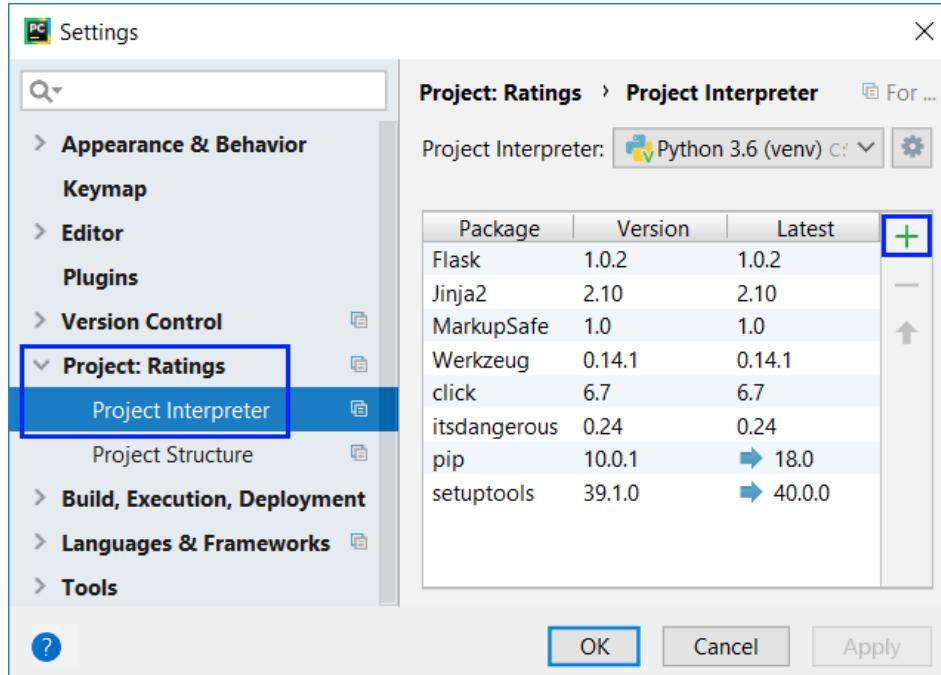
Започваме като създаваме нов проект в PyCharm от [File] -> [New Project] (или от началния прозорец):



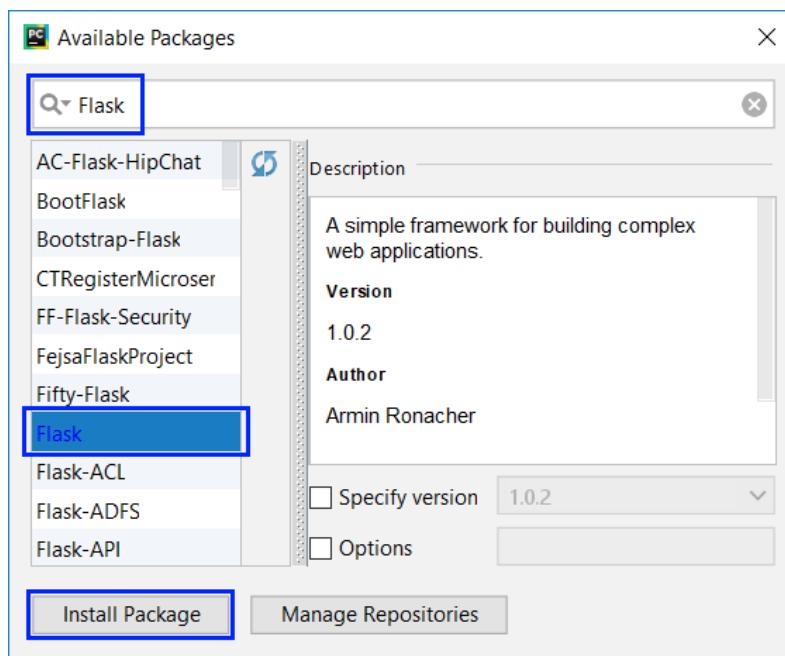
Даваме смислено име на проекта, например "Ratings". Избираме тип на текущия Python интерпретатор. Нека да е този по подразбиране:



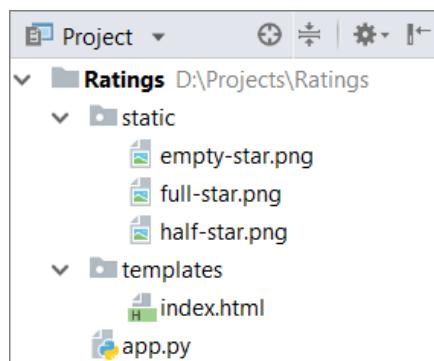
Ще използваме отново библиотеката Flask, която служи за създаване на уеб приложения. Преди да започнем да пишем код, трябва да инсталираме Flask. Нека си припомним как ставаше това. Отиваме в настройките на PyCharm [File] -> [Settings] -> [Project: Ratings] -> [Project Interpreter]. Там, натискаме бутона +:



Търсим **Flask** в прозореца, който излиза и натискаме бутона [Install Package]:



Сега добавяме **структурата** на проекта (файловете със заданието за този проект могат да бъдат свалени от <https://github.com/SoftUni/Programming-Basics-Book-Python-BG/tree/master/assets/chapter-6-1-assets>). Копираме ги от Windows Explorer и ги поставяме в папката на проекта Ratings с Copy/Paste:



За да заработи всичко, трябва да допищем кода. Първо отиваме във файла `index.html` (от папка `templates`) и търсим **TODO** секциите. На тяхно място въвеждаме следния код:

```

<h1>Rating</h1>
<form action="/DrawRating" method="GET">
    <input type="number" min="0"
           max="100" name="rating" value="{{rating}}"/>
    <input type="submit" value="Rate"/>
</form>
<br />
```

```
{% for star in range(full_stars) %}
    
{% endfor %}

{% for star in range(half_stars) %}
    
{% endfor %}

{% for star in range(empty_stars) %}
    
{% endfor %}
```

Горният код създава уеб форма **<form>** с едно поле **"rating"** за въвеждане на число в интервала [0 ... 100] и бутон **[Rate]** за изпращане на данните от формата към сървъра. След което, рисува с три отделни **for** цикъла съответния брой звездички - запълнени, полуупразни и празни.

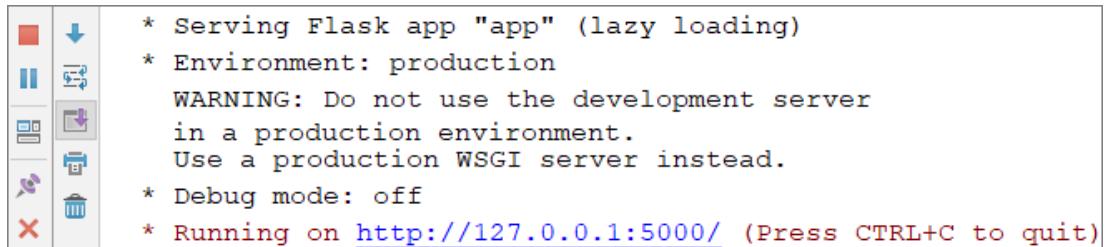
Действието, което ще обработи данните, се казва **/DrawRating**, което означава функция **draw\_rating()** във файла **app.py**:

```
1 from flask import Flask, render_template, redirect, request
2
3 app = Flask(__name__)
4
5 full_stars = 0
6 empty_stars = 10
7 half_stars = 0
8 rating = 0
9
10
11 @app.route('/DrawRating')
12 def draw_rating():
13     global rating
14     rating = int(request.args['rating'])
15     return calc_rating(rating)
```

Кодът от функцията **draw\_rating()** взима въведеното число **rating** от формата и го подава на функцията **calc\_rating(...)**. Функцията **calc\_rating(...)** извършва пресмятания и изчислява броя пълни звездички, броя празни звездички и броя половинки звездички, след което зарежда отново страницата, но вече с подадени нови стойности на променливите за звездичките. Имплементираме я по следния начин:

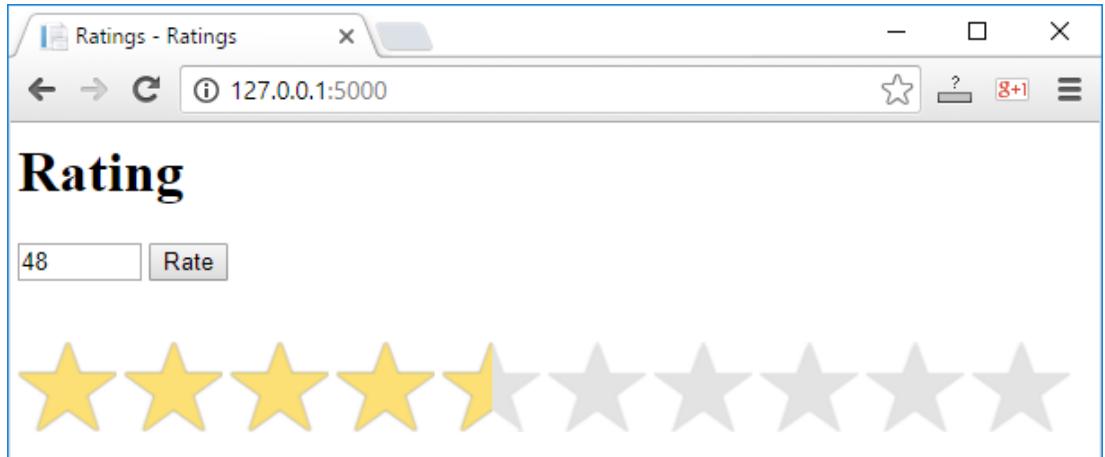
```
def calc_rating(rating):
    global full_stars
    full_stars = rating * 10 // 100
    global empty_stars
    empty_stars = (100 - rating) * 10 // 100
    global half_stars
    half_stars = 10 - full_stars - empty_stars
    return redirect('/')
```

Стартираме проекта с [Ctrl+Shift+F10] (или с [Десен бутон] → [Run 'app']) и изчакваме да се зареди:



```
* Serving Flask app "app" (lazy loading)
* Environment: production
WARNING: Do not use the development server
in a production environment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Отиваме на посочения адрес и се наслаждаваме на готовия проект:

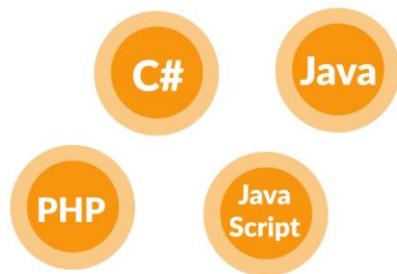


Ако имате проблеми с примерния проект по-горе, може да задавате въпроси във форума на СофтУни: <https://softuni.bg/forum>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 6.2. Вложени цикли – изпитни задачи

В предходната глава разгледахме **вложените цикли** и как да ги използваме за **рисуване** на различни **фигури** на конзолата. Научихме се как да отпечатваме фигури с различни размери, измисляйки подходяща логика на конструиране с използване на **единични и вложени for** цикли в комбинация с различни изчисления и програмна логика:

```
for r in range(5):
    print("*", end="")
    for c in range(5):
        print(" *", end="")
    print("")
```

Запознахме се и с **оператора \***, който дава възможност **даден стринг** да се печата определен от нас **брой** пъти:

```
print_me = ('abc' * 2)
```

## Изпитни задачи

Сега нека решим заедно няколко изпитни задачи, за да затвърдим наученото и да развием още алгоритмичното си мислене.

### Задача: чертане на крепост

Да се напише програма, която приема **цяло число n** и чертае **крепост** с ширина **2 \* n колони** и височина **n реда** като в примерите по-долу. Лявата и дясната колона във вътрешността си са широки **n / 2**.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	/^\^\^\\       \_/\_/\_	4	/^\^\^\\^\^\^\\         \_/\_/\_/\_	5	/^\^\^\\_\_\_\^^\^\\           \_/\_/\_/\_/\_

#### Входни данни

Входът на програмата се състои от един елемент (аргумент) - **цяло число n** в интервала [3 ... 1000].

## Изходни данни

Да се отпечатат на конзолата `n` текстови реда, изобразяващи **крепостта**, точно както в примерите.

## Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще се състоят само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3 ... 1000]. По тази причина ще конвертираме прочетената стрингова стойност към тип `int`:

```
n = int(input())
```

След като вече сме декларирали и инициализирали входните данни, ще разделим **крепостта** на три части:

- покрив
- тяло
- основа

От примерите можем да разберем, че **покривът** е съставен от **две кули** и **междинна част**. Всяка кула се състои от начало `/`, среда `^` и край `\`.

По условие лявата и дясната колона във вътрешността си са широки `n / 2`, следователно можем да отделим тази стойност в отделна **променлива**, като внимаваме, че ако като входни данни имаме **нечетно число**, при деление на две резултатът ще е реално число с цяла и дробна част. Тъй като в този случай ни трябва **само цялата част** (в условието на задачата виждаме, че при вход `3` броят на `^` във вътрешността на колоната е `1`, а при вход `5` е `3`), можем да я отделим с метода `math.trunc(...)` и да запазим само нейната стойност в новата ни променлива:

```
col_size = math.trunc(n / 2)
```



Добра практика е винаги, когато видим, че имаме израз, чиято стойност ще използваме **повече от един път**, да запазваме стойността му в променлива. Така от една страна, кодът ни ще бъде **по-лесно четим**, от друга страна, ще бъде **по-лесно да поправим евентуални грешки** в него, тъй като няма да се налага да търсим поотделно всяка употреба на израза.

Декларираме и втора **променлива**, в която ще пазим **стойността** на частта **между двете кули**. Знаем, че по условие общата ширина на крепостта е `n * 2`. Освен това имаме и две кули с по една наклонена черта за начало и край (общо 4 знака) и ширина **colSize**. Следователно, за да получим броя на знаците в междинната част, трябва да извадим размера на кулите от ширината на цялата крепост: `2 * n - 2 * colSize - 4`:

```
mid_size = (2 * n - 2 * colSize - 4) // 2
```

За да отпечатаме на конзолата покрива, ще използваме метода `format(...)` в комбинация с оператора `*`, който съединява даден символ в низ `n` на брой пъти:

```
print("/{0}\\\{1}/{0}\\\".format("^{n}" * col_size,
                                  "_" * mid_size))
```



\ е специален символ в езика Python и използвайки само него в метода `print(...)`, конзолата няма да го разпечата, затова с `\\` показваме на конзолата, че искаме да отпечатаме точно този символ, без да се интерпретира като специален (екранираме го, на английски се нарича “character escaping”).

Тялото на крепостта се състои от начало |, среда (**празно място**) и край |. Средата от празно място е с големина `2 * n - 2`. Броят на редовете за стени, можем да определим от дадените ни примери - `n - 3`:

```
for row in range(n - 3):
    print("|{0}|".format(...)))
```

За да нарисуваме предпоследния ред, който е част от основата, трябва да отпечатаме начало |, среда (**празно място**)\_(**празно място**) и край |. За да направим това, можем да използваме отново вече декларираните от нас променливи `col_size` и `mid_size`, защото от примерите виждаме, че са равни на броя `_` в покрива:

```
print("|{0}{1}{0}|".format(" " * (col_size + 1),
                           "_" * mid_size))
```

Добавяме към стойността на **празните места + 1**, защото в примерите имаме **едно** празно място повече.

Структурата на основата на крепостта е еднаква с тази на покрива. Съставена е от **две кули и междинна част**. Всяка една **кула** има начало \, среда \_ и край /:

```
print("\\{0}/{1}\\{0}/".format(..., ...))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1056#0>.

## Задача: пеперуда

Да се напише програма, която приема **цяло число n** и чертае **пеперуда** с ширина  $2 * n - 1$  колони и височина  $2 * (n - 2) + 1$  реда като в примерите по-долу. **Лявата и дясната ѝ част** са широки  $n - 1$ .

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	* \ /* @ */ \ *	5	*** \ / *** --- \ / --- *** \ / *** @ *** / \ *** --- / \ --- *** / \ ***	7	***** \ / ***** ----- \ / ----- ***** \ / ***** ----- \ / ----- ***** \ / ***** @ ***** / \ ***** ----- / \ ----- ***** / \ ***** ----- / \ ----- ***** / \ *****

## Входни данни

Входът се състои от един елемент (аргумент) - **цяло число n [3 ... 1000]**.

## Изходни данни

Да се отпечатат на конзолата  $2 * (n - 2) + 1$  текстови реда, изобразяващи пеперудата, точно както в примерите.

## Насоки и подсказки

Аналогично на предходната задача виждаме от условието, че **входните данни** ще се състоят само от едно **цяло число** в интервала **[3 ... 1000]**. По тази причина ще конвертираме прочетената стойност към тип **int**:

```
n =
```

Можем да разделим фигурата на 3 части - **горно крило, тяло и долно крило**. За да начертаем горното крило на пеперудата, трябва да го разделим на части - начало **\***, среда **\ /** и край **\***. След разглеждане на примерите можем да кажем, че горното крило на пеперудата е с големина **n - 2**:

```
half_row_size =
```

За да нарисуваме горното крило правим цикъл, който да се повтаря **half\_row\_size** пъти:

```
for i in range(half_row_size):
```

От примерите можем също така да забележим, че на **четен** ред имаме начало **\***, среда **\ /** и край **\***, а на **нечетен** - начало **-**, среда **\ /** и край **-**. Следователно при всяка итерация на цикъла трябва да направим **if-else** проверка дали редът, който печатаме, е четен или нечетен. От примерите, дадени в условието, виждаме, че броят на звездичките и тиретата на всеки ред също е равен на **n - 2**, т.

е. за тяхното отпечатване отново можем да използваме променливата **half\_row\_size**:

```
for i in range(half_row_size):
    if i % 2 == 1:
        print("{0}\\\\ /{0}"
              .format(" " * half_row_size))
    else:
        print("{0}\\\\ /{0}"
              .format(" " * half_row_size))
```

За да направим тялото на пеперудата е необходимо да отпечатаме на конзолата точно един ред. Структурата на тялото е с начало (**празно място**), среда @ и край (**празно място**). От примерите виждаме, че броят на празните места е **n - 1**:

```
print("{0}@{0}".format(" " * n - 1))
```

Остава да отпечатаме на конзолата и **долното крило**, което е аналогично на **горното крило**: единствено трябва да разменим местата на наклонените черти.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1056#1>.

### Задача: знак "Стоп"

Да се напише програма, която приема **цяло число n** и чертае **предупредителен знак STOP** с размери като в примерите по-долу.

### Примерен вход и изход

Вход	Изход	Вход	Изход
3	..... ...//____\\... ..//____\\.. .//____\\. //__STOP!__\\ \\____// .\\____//. .\\\\____//..	6	....._____. .....//____\\\. .... .....//____\\\. .... ....//____\\\. .... ...//____\\\. .... .//____\\\. .... //____STOP!____\\ \\____// .\\____//.. .\\\\____//... ...\\____//.... .....\\____//....

## Входни данни

Входът е един елемент (аргумент)- цяло число **n** в интервала [3 ... 1000].

## Изходни данни

Да се отпечатат на конзолата текстови редове, изобразяващи предупредителния знак STOP, точно както в примерите.

## Насоки и подсказки

Както и при предходните задачи, **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3 ... 1000]:

```
n = int(input())
```

Можем да разделим фигурата на **3 части** - горна, средна и долнна. **Горната част** се състои от две подчасти - начален ред и редове, в които знака се разширява. **Началния ред** е съставен от начало **..**, среда **\_** и край **..**. След разглеждане на примерите можем да кажем, че началото е с големина **n + 1** и е добре да отделим тази **стойност** в отделна **променлива**. Трябва да създадем и втора **променлива**, в която ще пазим **стойността на средата на началния ред** с големина **2 \* n + 1**:

```
dots = "."  
underscores = "_"
```

След като вече сме декларирали и инициализирали двете променливи, можем да отпечатаме на конзолата началния ред:

```
print("{0}{1}{0}" .format(dots, underscores, dots))
```

За да начертаем редовете, в които знака се "разширява", трябва да създадем **цикъл**, който да се завърти **n** брой пъти. Структурата на един ред се състои от начало **..**, **//** + среда **\_** + **\\"** и край **..**. За да можем да използваме отново създадените **променливи**, трябва да намалим **dots** с 1 и **underscores** с 2, защото ние вече сме **отпечатали** първия ред, а точките и долните черти в горната част от фигурата на всеки ред **намаляват**:

```
underscores  
dots -= 1
```

На всяка следваща итерация **началото** и **краят** намаляват с 1, а **средата** се увеличава с 2:

```
for i in range(n):  
    print("{0}//{1}\\".format(dots, underscores))  
  
    underscores += 2  
    dots -= 1
```

Средната част от фигурата има начало `// + _`, среда **STOP!** и край `_ + \\`. Броят на долните черти `_` е **(underscores - 5) / 2**:

```
print("//{0}STOP!{0}\\\\\\\"".format("".join(["_"] * underscores - 5) / 2)))
```

Долната част на фигурата, в която знака се **смаява**, можем да направим като отново създадем **цикъл**, който да се завърти **n** брой пъти. Структурата на един ред е начало `. + \\`, среда `_` и край `// + ..`. Броят на **точките** при първата итерация на цикъла трябва да е 0 и на всяка следваща да се **увеличава** с едно. Следователно можем да кажем, че броят на **точките в долната част от фигурата** е равен на **i**. За да работи нашата програма правилно, трябва на всяка итерация от **цикъла** да намаляваме броя на `_` с 2:

```
for i in range(n):
    print("{0}\\\\\\\"{1} //{0}".format("." * i,
                                         "_" * underscores))
    underscores -= 2
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1056#2>.

## Задача: стрелка

Да се напише програма, която приема **цяло нечетно число n** и чертае **вертикална стрелка** с размери като в примерите по-долу.

### Примерен вход и изход

Вход	Изход	Вход	Изход
3	<pre> .###. .#.# ##.## .#.# ..#.. </pre>		<pre> ..##### ..#....# ..#....# ..#....# ..#....# </pre>
		5	<pre> ##### .#....# ..#....# ..#....# ..#....# </pre>

### Входни данни

Входът се състои от **цяло нечетно число n** (аргумент) в интервала [3 ... 79].

## Изходни данни

Да се отпечатат на конзолата вертикална стрелка, при която "#" (диез) очертава стрелката, а "." - останалото.

## Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3 ... 1000]. По тази причина отново ще конвертираме прочетената стойност към тип **int**:

```
n = int(input())
```

Можем да разделим фигурата на **3 части** - горна, средна и долната. **Горната част** се състои от две подчасти - начален ред и тяло на стрелката. От примерите виждаме, че броят на **външните точки** в началния ред и в тялото на стрелката са **(n - 1) / 2**. Тази стойност можем да запишем в **променлива outer\_dots**:

```
outer_dots = int((n - 1) / 2)
```

Броят на **вътрешните точки** в тялото на стрелката е **(n - 2)**. Трябва да създадем променлива с име **inner\_dots**, която ще пази тази стойност:

```
inner_dots = n - 2
```

От примерите можем да видим структурата на началния ред. Трябва да използваме декларирани и инициализирани от нас **променливи outer\_dots** и **n**, за да отпечатаме **началния ред**:

```
print("{0}{1}{0}".format(".", " " * outer_dots, ))
```

За да нарисуваме на конзолата **тялото на стрелката**, трябва да създадем **цикъл**, който да се повтори **n - 2** пъти:

```
for i in range(n - 2):
    print("{0}#{1}#{0}".format(" " * outer_dots, ))
```

Средата на фигурата е съставена от начало **#**, среда **.** и край **#**. Броят на **#** е равен на **outer\_dots** и за това можем да използваме отново същата **променлива**:

```
print("{0}{1}{0}".format(" " * outer_dots, ))
```

За да начертаем **долната част на стрелката**, трябва да зададем нови стойности на двете **променливи outer\_dots** и **inner\_dots**.

```
outer_dots = 1
inner_dots = 2 * n - 5
```

При всяка итерация **outer\_dots** се увеличава с 1, а **inner\_dots** намалява с 2. Забелязваме, че тъй като на предпоследния ред стойността на **inner\_dots** ще е 1 и при последвала итерация на цикъла тя ще стане **отрицателно число**. Тъй като оператора **\*** не може да съедини символ 0 или отрицателен брой пъти в низ, няма да се изведе нищо на конзолата. Един вариант да избегнем това е да отпечатаме последния ред на фигурата отделно. Височината на долната част на стрелката е **n - 1**, следователно **цикълът**, който ще отпечатва всички редове без последния, трябва да се завърти **n - 2** пъти:

```
for i in range( n - 1 ) :
    print( " " * outer_dots + "*" * inner_dots + " " * outer_dots )
    outer_dots += 1
    inner_dots -= 2
```

Последният ред от нашата фигура е съставен от начало **.**, среда **#** и край **.**. Броят на **.** е равен на **outer\_dots**:

```
print("{0}#{0}.format("." * outer_dots))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1056#3>.

## Задача: брадва

Да се напише програма, която приема **цяло число n** и чертае брадва с размери, показани по-долу. Ширината на брадвата е **5 \* n** колони.

### Примерен вход и изход

Вход	Изход	Вход	Изход
2	<pre>-----**-- -----*-* *****-* -----***-</pre>	5	<pre>-----**-- -----*-* -----*-* -----*-* -----*-* *****-* *****-* -----*-* -----*****</pre>

### Входни данни

Входът се състои от един елемент (аргумент) - **цяло число n** в интервала [2..42].

## Изходни данни

Да се отпечата на конзолата **брадва**, точно както е в примерите.

## Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [2 ... 42]. По тази причина ще използваме тип **int**. След което, за решението на задачата е нужно първо да изчислим големината на **тиритата от ляво**, **средните тирета**, **тиритата от дясно** и цялата дължина на фигуранта:

```
n = int(input())
width = 5 * n
left_dashes = 3 * n
middle_dashes = 0
right_dashes = width - left_dashes - middle_dashes - 2
```

След като сме декларирали и инициализирали променливите, можем да започнем да изчертаваме фигуранта като започнем с **горната част**. От примерите можем да разберем каква е структурата на **първия ред** и да създадем цикъл, който се повтаря **n** на брой пъти. При всяка итерация от цикъла **средните тирета** се увеличават с 1, а **тиритата от дясно** се намаляват с 1:

```
for i in range(n):
    print("{0}*{1}*{2}".format("-" * left_dashes,
                                "-" * middle_dashes,
                                "-" * right_dashes))
```



Сега следва да нарисуваме **дръжката на брадвата**. За да можем да използваме отново създадените **променливи** при чертането на дръжката на брадвата, трябва да намалим **средните тирета** с 1, а **тези от дясно и отляво** да увеличим с 1.

```
middle_dashes -= 1
right_dashes += 1
left_dashes += 1
```

**Дръжката на брадвата** можем да нарисуваме, като завъртим цикъл, който се повтаря

**n / 2** пъти. Можем да отделим тази стойност в отделна **променлива**, като внимаваме, че ако като входни данни имаме **нечетно число**, при деление на 2 резултатът ще е **реално число** с цяла и дробна част. Тъй като в този случай ни трябва **само цялата част** (от условието на задачата виждаме, че при вход **5** височината на дръжката на брадвата е **2**), можем да използваме метода **math.trunc(...)**, с който да запазим само нейната стойност в новата ни

променлива **axe\_height**. От примерите можем да разберем, каква е структурата на дръжката:

```
axe_height = int(n / 2)

for i in range(axe_height):
    print("{0}{1}*{2}".format("*" * left_dashes,
                             "--" * middle_dashes,
                             "--" * right_dashes))
```

Долната част на фигурата, трябва да разделим на две подчасти - **глава на брадвата** и **последния ред от фигурата**. Главата на брадвата ще отпечатаме на конзолата, като направим цикъл, който да се повтаря **axe\_height - 1** пъти. На всяка итерация **тиретата отляво** и **тиретата отдясно** намаляват с 1, а **средните тирета** се увеличават с 2:

```
left_dashes -= 1
```

```
middle_dashes += 2
left_dashes -= 1
right_dashes -= 1
```

За **последния ред** от фигурата, можем отново да използваме трите, вече декларириани и инициализирани променливи **left\_dashes**, **middle\_dashes**, **right\_dashes**.

```
print("{0}*{1}*{2}".format("--" * left_dashes,
                           "*" * middle_dashes,
                           "--" * right_dashes))
```

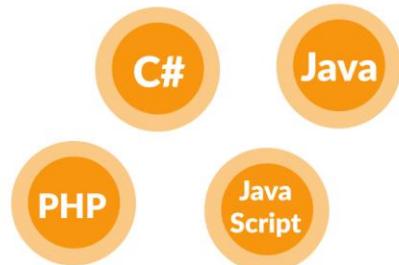
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1056#4>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтуни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтуни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 7.1. По-сложни цикли

След като научихме какво представляват и за какво служат **for** циклите, сега предстои да се запознаем с **други видове цикли**, както и с някои **по-сложни конструкции за цикъл**. Те ще разширят познанията ни и ще ни помагат в решаването на по-трудни и по-предизвикателни задачи. По-конкретно, ще разгледаме как се ползват следните програмни конструкции:

- цикли **със стъпка**
- **while** цикли
- **while + break** цикли
- **безкрайни** цикли

В настоящата тема ще разберем и какво представлява операторът **break**, както и **как** чрез него да **прекъснем** един цикъл. Ще се научим да следим за **грешки** по време на изпълнението на програмата, използвайки **try-except** конструкцията.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=IPJigJNDuKQ>.

## Цикли със стъпка

В главата "[Повторения \(цикли\)](#)" научихме как работи **for** цикълът и вече знаем кога и с каква цел да го използваме. В тази тема ще обърнем **внимание** на една определена и много важна **част от конструкцията** му, а именно **стъпката**.

### Какво представлява стъпката?

Стъпката е тази **част** от конструкцията на **range** функцията, която указва с **колко** да се **увеличи** или **намали** стойността на **водещата** му променлива. Тя се декларира последна в скелета на **range** функцията.

По подразбиране е с **размер 1** и не се добавя в **range** функцията. Ако искаме стъпката ни да е **различна от 1**, при писане на **range** функцията, добавяме още едно число, което е нашата стъпка. При стъпка **10**, цикълът би изглеждал по следния начин:

```
n = int(input())
for i in range(1, n + 1, 10):
    print(i)
```

Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **стъпката** във **for** цикъл.

### Пример: числата от 1 до N през 3

Да се напише програма, която отпечатва числата от 1 до n със **стъпка 3**. Например, ако **n = 100**, то резултатът ще е: 1, 4, 7, 10, ..., 94, 97, 100.

Можем да решим задачата чрез следната поредица от действия (алгоритъм):

- Четем числото **n** от входа на конзолата.
- Изпълняваме **for цикъл** от **1** до **n** (включително и **n**) с размер на стъпката **3**.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
n = int(input())
for i in range(1, n + 1, 3):
    print(i)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#0>.

## Пример: числата от N до 1 в обратен ред

Да се напише програма, която отпечатва числата **от n до 1 в обратен ред** (стъпка **-1**). Например, ако **n = 100**, то резултатът ще е: **100, 99, 98, ..., 3, 2, 1**.

Можем да решим задачата по следния начин:

- Четем числото **n** от входа на конзолата.
- Създаваме **for цикъл**, от **n** до **0**.
- Дефинираме размера на стъпката: **-1**.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
n = int(input())
for i in range(n, 0, -1):
    print(i)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#1>.

## Пример: числата от 1 до $2^n$ с for цикъл

В следващия пример ще разгледаме ползването на обичайната стъпка **1**.

Да се напише програма, която отпечатва числата **от 1 до  $2^n$**  (две на степен **n**). Например, ако **n = 10**, то резултатът ще е **1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024**.

```
n = int(input())
num = 1
for i in range(0, n + 1):
    print(num)
    num = num * 2
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#2>.

## Пример: четни степени на 2

Да се отпечатат четните степени на 2 до  $2^n$ :  $2^0, 2^2, 2^4, 2^8, \dots, 2^n$ . Например, ако  $n = 10$ , то резултатът ще е 1, 4, 16, 64, 256, 1024.

Ето как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална **стойност 1**.
- За **стъпка** на цикъла задаваме стойност **2**.
- В **тялото на цикъла**: отпечатваме стойността на текущото число и **увеличаваме текущото число num 4 пъти** (според условието на задачата).

```
n = int(input())
num = 1
for i in range(0, n + 1, 2):
    print(num)
    num = num * 2 * 2
```

## Тестване в Judge системата

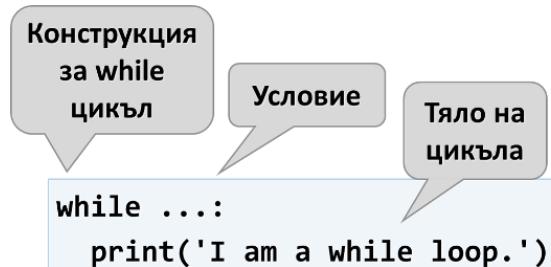
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#3>.

## While цикъл

Следващият вид цикли, с които ще се запознаем, се наричат **while** цикли. Специфичното при тях е, че повтарят блок от команди, докато дадено условие е истина. Като структура се различават от тази на **for** циклите, даже имат опростен синтаксис.

### Какво представлява while цикълът?

В програмирането **while** цикълът се използва, когато искаме да **повтаряме** извършването на определена логика, докато **е в сила дадено условие**. Под "условие", разбираме всеки израз, който връща **true** или **false**. Когато условието стане **грешно**, **while** цикълът прекъсва изпълнението си и програмата **продължава** с изпълняването на кода след цикъла. Конструкцията за **while** цикъл изглежда по този начин:



Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **while** цикъла.

## Пример: редица числа $2k+1$

Да се напише програма, която отпечатва всички **числа  $\leq n$**  от редицата: **1, 3, 7, 15, 31, ...**, като приемем, че всяко следващо число = **предишно число \* 2 + 1**.

Ето как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална **стойност 1**.
- За условие на цикъла слагаме **текущото число  $\leq n$** .
- В **тялото на цикъла**: отпечатваме стойността на текущото число и увеличаваме текущото число, използвайки формулата от условието на задачата.

Ето и примерна реализация на описаната идея:

```
n = int(input())
num = 1
while num <= n:
    print(num)
    num = 2 * num + 1
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#4>.

## Пример: число в диапазона [1 ... 100]

Да се въведе цяло число в диапазона **[1 ... 100]**. Ако то е невалидно, да се въведе отново. В случая, за невалидно число ще считаме всяко такова, което **не е** в зададения диапазон.

За да решим задачата, можем да използваме следния алгоритъм:

- Създаваме променлива **num**, на която присвояваме целочислената стойност, получена от входа на конзолата.
- За условие на цикъла слагаме израз, който е **True**, ако числото от входа **не е** в диапазона посочен в условието.
- В **тялото на цикъла**: отпечатваме съобщение със съдържание "**Invalid number!**" на конзолата, след което присвояваме нова стойност за **num** от входа на конзолата.
- След като вече сме валидирали въведеното число, извън тялото на цикъла отпечатваме стойността му.

Ето и примерна реализация на алгоритъма чрез **while** цикъл:

```

num = int(input())
while num < 1 or num > 100:
    print('Invalid number!')
    num = int(input())
print('The number is: %d' % num)

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#5>.

## Най-голям общ делител (НОД)

Преди да продължим към следващата задача, е необходимо да се запознаем с определението за **най-голям общ делител (НОД)**.

**Определение за НОД:** най-голям общ делител на две **естествени** числа **a** и **b** е най-голямото число, което се дели **едновременно** и на **a**, и на **b** без остатък.

Например:

a	b	НОД
24	16	8
67	18	1
12	24	12

a	b	НОД
15	9	3
10	10	10
100	88	4

## Алгоритъм на Евклид

В следващата задача ще използваме един от първите публикувани алгоритми за намиране на НОД - **алгоритъм на Евклид**:

Докато не достигнем остатък 0:

- Делим по-голямото число на по-малкото.
- Вземаме остатъка от делението.

Псевдо-код за алгоритъма на Евклид:

```

while b ≠ 0
    oldB = b
    b = a % b
    a = oldB
print a

```

## Пример: най-голям общ делител (НОД)

Да се подадат **цели** числа **a** и **b** и да се намери **НОД(a, b)**.

Ще решим задачата чрез **алгоритъма на Евклид**:

- Създаваме променливи **a** и **b**, на които присвояваме **целочислени** стойности, взети от входа на конзолата.
- За условие на цикъла слагаме израз, който е **True**, ако числото **b** е различно от 0.
- В **тялото на цикъла** следваме указанията от псевдо кода:
  - Създаваме временна променлива, на която присвояваме **текущата** стойност на **b**.
  - Присвояваме нова стойност на **b**, която е остатъка от делението на **a** и **b**.
  - На променливата **a** присвояваме **предишната** стойност на променливата **b**.

- След като цикълът приключи и сме установили НОД, го отпечатваме на экрана.

```
a = int(input())
b = int(input())

while b != 0:
    old_b = b
    b = a % b
    a = old_b
print('GCD = %d' % a)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#6>.

## While True + break цикъл

Следващият тип конструкция за цикъл, с която ще се запознаем докато изучаваме програмирането, е **while True + break** цикълът (за по-кратко **while + break** цикъл). Неговата идея е да повтаря фрагмент от кода многократно докато не се достигне до изрично прекратяване на цикъла, обикновено след **if** проверка в тялото на цикъла. Ето как изглежда този цикъл на практика като Python код:

```
while True:
    print('I am a do...while loop.')
    if not ...:
        break
```

Горният пример се нарича “**обърнат while цикъл**”, защото условието за изход от цикъла е **накрая**, а не в началото. По същина горната конструкция представлява “**безкраен цикъл**” с проверка на дадено условие за изход вътре в тялото на цикъла.

В програмирането операторът **break** безусловно прекратява даден цикъл и преминава към първата инструкция веднага след него. В горния пример в края на цикъла се проверява дадено условие и ако то не е вярно, цикълът се прекратява.

Конструкцията за цикъл **while + break** в много други езици за програмиране се реализира с **do-while** конструкция (в превод “прави-докато”), но последната няма директен еквивалент в Python. За да се постигне същото поведение, в Python се използва безкраен цикъл (**while True**) и когато се достигне условието за изход от него, цикълът се прекъсва (с **break**).

Конструкцията **while + break** предоставя повече гъвкавост, отколкото **while** циклите, защото позволява изходът от цикъла да е на **произволно място** в него (например в началото, в средата или в края), дори позволява да има изход от цикъла на няколко различни места (с няколко **break** оператора).

По структура **while + break** цикълът наподобява много класическия **while** цикъл, но има съществена разлика: **while** се изпълнява 0 или повече пъти (според входното условие на цикъла), докато **while + break** изпълнява тялото си **поне веднъж**. Защо се случва това? В конструкцията на **while True + break** цикъла, **условието** винаги се проверява **вътре в тялото му**, докато при класическият **while** цикъл проверката за изход от цикъла е винаги в началото, преди неговото тяло.

След като се запознахме с концепцията за **while + break** цикъл с условие за изход, което не е задължително в началото, сега нека преминем през обичайната поредица от примерни задачи, в които можем да приложим наученото.

## Пример: изчисляване на факториел

За естествено число **n** да се изчисли  $n! = 1 * 2 * 3 * \dots * n$ . Например, ако  $n = 5$ , то резултатът ще бъде:  $5! = 1 * 2 * 3 * 4 * 5 = 120$ .

Ето как по-конкретно можем да пресметнем факториел:

- Създаваме променливата **n**, на която присвояваме целочислена стойност взета от входа на конзолата.
- Създаваме още една променлива - **fact**, чиято начална стойност е 1. Ней ще използваме за изчислението и съхранението на факториела.
- За условие на цикъла ще използваме **n > 1**, тъй като всеки път, когато извършим изчисленията в тялото на цикъла, ще намаляваме стойността на **n** с 1.
- В тялото на цикъла:
  - Присвояваме нова стойност на **fact**, която е резултат от умножението на текущата стойност на **fact** с текущата стойност на **n**.
  - Намаляваме стойността на **n** с -1.
- Извън тялото на цикъла отпечатваме крайната стойност на факториела.

```

n = int(input())
fact = 1
while True:
    fact = fact * n
    n -= 1
    if not n > 1:
        break

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#7>.

## Пример: сумиране на цифрите на число

Да се сумират цифрите на цяло положително число **n**. Например, ако **n** = 5634, то резултатът ще бъде:  $5 + 6 + 3 + 4 = 18$ .

Можем да използваме следната идея, за да решим задачата:

- Създаваме променливата **n**, на която присвояваме стойност, равна на въведеното от потребителя число.
- Създаваме втора променлива **sum**, чиято начална стойност е 0. Ней ще използваме за изчислението и съхранението на резултата.
- За условие на цикъла ще използваме **n > 0**, тъй като след всяко изчисление на резултата в тялото на цикъла, ще премахваме последната цифра от **n**.
- В тялото на цикъла:
  - Присвояваме нова стойност на **sum**, която е резултат от събирането на текущата стойност на **sum** с последната цифра на **n**.
  - Присвояваме нова стойност на **n**, която е резултат от премахването на последната цифра от **n**.
- Извън тялото на цикъла отпечатваме крайната стойност на сумата.

Ето и примерна реализация на описаната идея:

```

n = int(input())
sum = 0

while True:
    sum = sum + (n % 10)
    n = n // 10
    if not n > 0:
        break
print('Sum of digits: %d' % sum)

```



- `n % 10`: връща последната цифра на числото `n`.
- `n // 10`: изтрива последната цифра на `n`.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#8>.

## Безкрайни цикли и операторът `break`

До момента се запознахме с различни видове цикли, като научихме какви конструкции имат те и как се прилагат. Следва да разберем какво е **безкраен цикъл**, кога възниква и как можем да прекъснем изпълнението му чрез оператора **`break`**.

### Безкраен цикъл. Що е то?

Безкраен цикъл наричаме този цикъл, който **повтаря безкрайно** изпълнението на тялото си. При безкрайните **`while`** цикли проверката за край е условен израз, който **винаги** връща **`true`**. Ето как изглежда **безкраен `while` цикъл**:

```
while True:
    print('Infinite loop')
```

### Оператор `break`

Вече знаем, че безкрайният цикъл изпълнява определен код до безкрайност, но какво става, ако желаем в определен момент при дадено условие, да излезем принудително от цикъла? На помощ идва операторът **`break`**, в превод - **спри, прекъсни**.



Операторът **`break`** спира изпълнението на цикъла към момента, в който е извикан, и продължава от първия ред след края на цикъла. Това означава, че текущата итерация на цикъла няма да бъде завършена до край и съответно останалата част от кода в тялото на цикъла няма да се изпълни.

### Пример: прости числа

В следващата задача се изиска да направим **проверка за просто число**. Преди да продължим към нея, нека си припомним какво са простите числа.

**Определение:** едно цяло число е **просто**, ако се дели без остатък единствено на себе си и на 1. По дефиниция простите числа са положителни и по-големи от 1. Най-малкото просто число е **2**.

Можем да приемем, че едно цяло число `n` е просто, ако  $n > 1$  и `n` не се дели на число между 2 и `n-1`.

Първите няколко прости числа са: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ...

За разлика от тях, **непростите (композитни) числа** са такива числа, чиято композиция е съставена от произведение на прости числа.

Ето няколко примерни непрости числа:

- $10 = 2 * 5$
- $42 = 2 * 3 * 7$
- $143 = 13 * 11$

**Алгоритъм за проверка** дали дадено цяло число е **просто**: проверяваме дали  $n > 1$  и дали  $n$  се дели на  $2, 3, \dots, n-1$  без остатък.

- Ако се раздели на някое от числата, значи е **композитно**.
- Ако не се раздели на никое от числата, значи е **просто**.



Можем да оптимизираме алгоритъма, като вместо проверката да е до  $n-1$ , да се проверяват делителите до  $\sqrt{n}$ . Помислете защо.

## Пример: проверка за просто число. Оператор `break`

Да се провери дали едно число  $n$  е просто. Това ще направим като проверим дали  $n$  се дели на числата между  $2$  и  $\sqrt{n}$ .

Ето го алгоритъма за проверка за просто число, разписан постъпково:

- Създаваме променливата `n`, на която присвояваме цяло число въведено от входа на конзолата.
- Създаваме булева променлива `is_prime` с начална стойност `True`. Приемаме, че едно число е просто до доказване на противното.
- Създаваме `for` цикъл, на който като начална стойност за променливата на цикъла задаваме  $2$ , за условие `текущата ѝ стойност  $\leq \sqrt{n}$` . Стъпката на цикъла е  $1$ .
- В **тялото на цикъла** проверяваме дали `n`, разделено на `текущата стойност` има остатък. Ако от делението **няма остатък**, то променяме `is_prime` на `False` и излизаме принудително от цикъла чрез оператор `break`.
- В зависимост от стойността на `is_prime` отпечатваме дали числото е просто (`True`) или съответно (`False`).

Ето и примерна имплементация на описания алгоритъм:

```
import math

n = int(input())
prime = True
```

```

for i in range(2, int(math.sqrt(n)) + 1):
    if n % i == 0:
        prime = False
        break

if prime:
    print('Prime')
else:
    print('Not prime')

```

Оставаме да добавите проверка дали входното число е по-голямо от 1, защото по дефиниция числа като 0, 1, -1 и -2 не са прости.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1057#9>.

## Пример: оператор break в безкраен цикъл

Да се напише програма, която проверява дали едно число **n** е четно, ако е - да се отпечатва на екрана. За четно считаме число, което се дели на 2 без остатък. При невалидно число да се връща към повторно въвеждане и да се изписва съобщение, което известява, че въведеното число не е четно.

Ето една идея как можем да решим задачата:

- Създаваме променлива **n**, на която присвояваме начална стойност 0.
- Създаваме безкраен **while** цикъл, като за условие ще зададем **True**.
- В тялото на цикъла:
  - Вземаме целочислена стойност от входа на конзолата и я присвояваме на **n**.
  - Ако **числото е четно**, излизаме от цикъла чрез **break**.
  - В **противен случай** извеждаме съобщение, което гласи, че **числото не е четно**. Итерациите продължават, докато не се въведе четно число.
- Отпечатваме четното число на екрана.

Ето и примерна имплементация на идеята:

```

while True:
    print('Enter even number: ', end=' ')
    n = int(input())
    if n % 2 == 0:
        break
    print('The number is not even.')
print('Even number entered: %d' % n)

```

Забележка: макар кодът по-горе да е коректен, той няма да работи, ако вместо число потребителят въведе текст, например "Invalid number". Тогава парсването на текста към число ще се счупи и програмата ще покаже **съобщение за грешка (изключение)**. Как да се справим с този проблем и как да прихващаме и обработваме изключения чрез **try-except** конструкцията ще научим след малко.

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#10>.

## Вложени цикли и операторът break

След като вече научихме какво са **вложените цикли** и как работи операторът **break**, е време да разберем как работят двете заедно. За по-добро разбиране, нека стъпка по стъпка да напишем **програма**, която трябва да направи всички възможни комбинации от **двойки** числа. Първото число от комбинацията е нарастващо от 1 до 3, а второто е намаляващо от 3 до 1. Задачата трябва да продължи изпълнението си, докато **i + j** не е равно на 2 (т.e. **i = 1** и **j = 1**).

Желаният резултат е:

```
Run: python_file
"C:\Program Files (x86)\Python\3.6.0\python.exe"
1 3
1 2
```

Ето едно **грешно решение**, което изглежда правилно на пръв поглед:

```
for i in range(1, 4):
    for j in range(3, 0, -1):
        if i + j == 2:
            break
        print('%d %d' % (i, j))
```

Ако оставим програмата ни по този начин, резултатът ни ще е следният:

```
Run: python_file
"C:\Program Files (x86)\Python\3.6.0\python.exe"
1 3
1 2
2 3
2 2
2 1
3 3
3 2
3 1
Process finished with exit code 0
```

Защо се получава така? Както виждаме, в резултата липсва "1 1". Когато програмата стига до там, че **i = 1 и j = 1**, тя влиза в **if** проверката и изпълнява **break** операцията. По този начин се **излиза от вътрешния цикъл**, но след това продължава изпълнението на външния. **i** нараства, програмата влиза във вътрешния цикъл и принтира резултата.



Когато във **вложен цикъл** използваме оператора **break**, той прекъсва изпълнението **само** на вътрешния цикъл.

Какво е **правилното решение**? Един начин за решаването на този проблем е чрез деклариране на **bool** променлива, която следи за това, дали трябва да продължава въртенето на цикъла. При нужда от изход (излизане от всички вложени цикли), се променя стойността на променливата на **True** и се излиза от вътрешния цикъл с **break**, а при последваща проверка се напуска и външния цикъл. Ето и примерна имплементация на тази идея:

```
has_to_end = False
for i in range(1, 4):
    if has_to_end == False:
        for j in range(3, 0, -1):
            if i + j == 2:
                has_to_end = True
                break
    print('%d %d' % (i, j))
```

По този начин, когато **i + j = 2**, програмата ще направи променливата **has\_to\_end = True** и ще излезе от вътрешния цикъл. При следващото завъртане на външния цикъл, чрез **if** проверката, програмата няма да може да стигне до вътрешния цикъл и ще прекъсне изпълнението си.

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#11>.

## Справяне с изключения: try-except

Последното, с което ще се запознаем в тази глава, е как да "улавяме" **грешки** (**exceptions**) чрез конструкцията **try-except**.

### Какво е try-except?

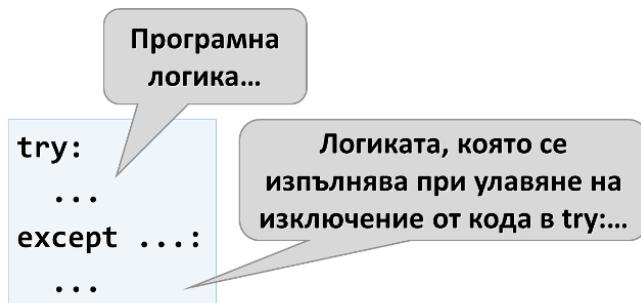
Програмната конструкция **try-except** служи за **прихващане и обработка на изключения (грешки)** по време на изпълнението на програмата.

В програмирането **изключенията (exceptions)** представляват уведомление за дадено събитие, което наруши нормалната работа на една програма. Такива

събития **прекъсват изпълнението** на програмата и тя търси кой да обработи настъпилата ситуация. Ако не намери, изключението се отпечатва на конзолата (т.е. програмата "гърми"). Ако намери, **изключението се обработва** и програмата продължава нормалното си изпълнение, без да "гърми". След малко ще видим как точно става това.

## Конструкция на try-except

Конструкцията **try-except** има различни варианти, но за сега ще се запознаем само с най-основния от тях:



В следващата задача ще видим нагледно, как да се справим в ситуация, в която потребителят въвежда вход, различен от число (например **string** вместо **int**), чрез **try-except**.

### Пример: справяне с невалидни числа чрез try-except

Да се напише програма, която проверява дали едно число **n** е четно и ако е, да се отпечатва на екрана. При **невалидно въведено** число да се изписва съобщение, че въведенния вход не е валидно число и въвеждането да продължи отново.

Ето как можем да решим задачата:

- Създаваме безкраен **while** цикъл, като за условие ще зададем **True**.
- В тялото на цикъла:
  - Създаваме **try-except** конструкция.
  - В **try** блока пишем програмната логика за четене на потребителския вход, парсването му до число и проверката за четност.
  - При **четно число** го отпечатваме и излизаме от цикъла (с **break**). Програмата си е свършила работата и приключва.
  - При **нечетно число** отпечатваме съобщение, че се изисква четно число, без да излизаме от цикъла (защото искаме той да се повтори отново).
  - Ако **хванем изключение** при изпълнението на **try** блока, изписваме съобщение за невалидно въведено число (и цикълът съответно се повтаря, защото не излизаме изрично от него).

Ето и примерна имплементация на описаната идея:

```
while True:
    try:
        print('Enter even number: ', end=' ')
        n = int(input())
        if n % 2 == 0:
            print('Even number entered: %d' % n)
            break
    except ValueError:
        print('Invalid number.')
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#10>.

Сега вече решението трябва да работи винаги: независимо дали въвеждаме цели числа, невалидни числа (например твърде много цифри) или текст, които не съдържат числа.

## Задачи с цикли

В тази глава се запознахме с няколко нови вида цикли, с които могат да се правят повторения с по-сложна програмна логика. Да решим няколко задачи, използвайки новите знания.

### Задача: числа на Фиbonачи

Числата на Фиbonачи в математиката образуват редица, която изглежда по следния начин: 1, 1, 2, 3, 5, 8, 13, 21, 34, ....

Формулата за образуване на редицата е:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

### Примерен вход и изход

Вход (n)	Изход	Коментар
10	89	$F(11) = F(9) + F(8)$
5	8	$F(5) = F(4) + F(3)$
20	10946	$F(20) = F(19) + F(18)$

Вход (n)	Изход
0	1
1	1

Да се въведе **цяло** число **n** и да се пресметне **n**-тото число на Фиbonачи.

## Насоки и подсказки

Идея за решаване на задачата:

- Създаваме променлива **n**, на която присвояваме целочислена стойност от входа на конзолата.
- Създаваме променливите **f0** и **f1**, на които присвояваме стойност **1**, тъй като така започва редицата.
- Създаваме **for** цикъл от нула до крайна стойност **n - 1**.
- В тялото на цикъла:
  - Създаваме временна променлива **f\_next**, на която присвояваме следващото число в редицата на Фиbonacci.
  - На **f0** присвояваме текущата стойност на **f1**.
  - На **f1** присвояваме стойността на временната променлива **f\_next**.
- Извън цикъла отпечатваме числото n-тото число на Фиbonacci.

Примерна имплементация:

```
n = int(input())
f0 = 1
f1 = 1
for i in range(0, n - 1):
    f_next = f0 + f1
    f0 = f1
    f1 = f_next
print(f1)
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#12>.

## Задача: пирамида от числа

Да се отпечатат числата 1 ... n в пирамида като в примерите по долу. На първия ред печатаме едно число, на втория ред печатаме две числа, на третия ред печатаме три числа и т.н. докато числата свършат. На последния ред печатаме толкова числа, колкото останат докато стигнем до n.

### Примерен вход и изход

Вход	Изход
7	1 2 3 4 5 6 7

Вход	Изход
5	1 2 3 4 5

Вход	Изход
10	1 2 3 4 5 6 7 8 9 10

## Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** (по редове и колони) с печатане в тях и излизане при достигане на последното число. Ето идеята, разписана по-подробно:

- Създаваме променлива **n**, на която присвояваме целочислена стойност, прочетена от конзолата.
- Създаваме променлива **num** с начална стойност 1. Тя ще пази броя на отпечатаните числа. При всяка итерация ще я **увеличаваме** с **1** и ще я принтираме.
- Създаваме **външен for** цикъл, който ще отговаря за **редовете** в таблицата. Наименуваме променливата на цикъла **row** и ѝ задаваме начална стойност 1. За крайна стойност слагаме **n + 1**.
- В тялото на цикъла създаваме **вътрешен for** цикъл, който ще отговаря за **колоните** в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална стойност 1. За условие слагаме **row + 1** (**row** = брой цифри на ред).
- В тялото на вложния цикъл:
  - Проверяваме дали **col > 1**, ако да – принтираме разстояние. Ако не направим тази проверка, а директно принтираме разстоянието, ще имаме ненужно такова в началото на всеки ред.
  - **Отпечатваме** числото **num** в текущата клетка на таблицата и го **увеличаваме с 1**.
  - Правим проверка за **num > n**. Ако **num** е по-голямо от **n**, прекъсваме въртенето на **вътрешния цикъл**.
- Отпечатваме **празен ред**, за да преминем на следващия.
- Отново проверяваме дали **num > n**. Ако е по-голямо, прекъсваме изпълнението на програмата ни чрез **break**.

Ето и примерна имплементация:

```
n = int(input())
num = 1

for row in range(1, n + 1):
    for col in range(1, row + 1):
        if col > 1:
            print(' ', end='')
        print(num, end=' ')
        num += 1
        if num > n:
            break
    print()
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#13>.

### Задача: таблица с числа

Да се отпечатат числата  $1 \dots n$  в таблица като в примерите по-долу.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
3	1 2 3 2 3 2 3 2 1	4	1 2 3 4 2 3 4 3 3 4 3 2 4 3 2 1

#### Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** и малко изчисления в тях:

- Четем от конзолата размера на таблицата в целочислена променлива **n**.
- Създаваме **for** цикъл, който ще отговаря за редовете в таблицата. Наименуваме променливата на цикъла **row** и ѝ задаваме начална **стойност 0**. За краяна стойност слагаме **n**. Размерът на стъпката е **1**.
- В **тялото на цикъла** създаваме вложен **for** цикъл, който ще отговаря за колоните в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална **стойност 0**. За краяна стойност слагаме **n**. Размерът на стъпката е **1**.
- В **тялото на вложения цикъл**:
  - Създаваме променлива **num**, на която присвояваме резултата от **текущият ред + текущата колона + 1** (+1, тъй като започваме броенето от 0).
  - Правим проверка за **num > n**. Ако **num** е **по-голямо** от **n**, присвояваме нова стойност на **num** равна на **два пъти n - текущата стойност за num**. **Това правим с цел да не превишаваме n\*\*** в никоя от клетките на таблицата.
    - Отпечатваме числото от текущата клетка на таблицата.
- Отпечатваме **празен ред** във външния цикъл, за да преминем на следващия ред.

Ето и примерна имплементация на описаната идея:

```

n = int(input())
for row in range(n):
    for col in range(n):
        num = row + col + 1
        if num > n:
            num = 2 * n - num
        print('%d ' % num, end='')
print()

```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1057#14>.

## Какво научихме от тази глава?

Можем да използваме **for** цикли със стъпка:

```

for i in range(1, n + 1, 3):
    print(i)

```

Циклите **while** / **while** + **break** се повтарят докато е в сила дадено **условие**:

```

num = 1
while num <= n:
    print(num)
    num += 1

```

Ако се наложи да прекъснем изпълнението на цикъл, го правим с оператора **break**:

```

n = 0
while True:
    n = int(input())
    if n % 2 == 0:
        break # even number -> exit from the loop
    print("The number is not even.")
print("Even number entered: {}".format(n))

```

Вече знаем как да прихващаме **грешки** по време на изпълнение:

```

try:
    print("Enter even number: ", end="")
    n = int(input())
except ValueError:
    print("Invalid number.")
# Ако int(...) грямне, ще се изпълни except блокът

```

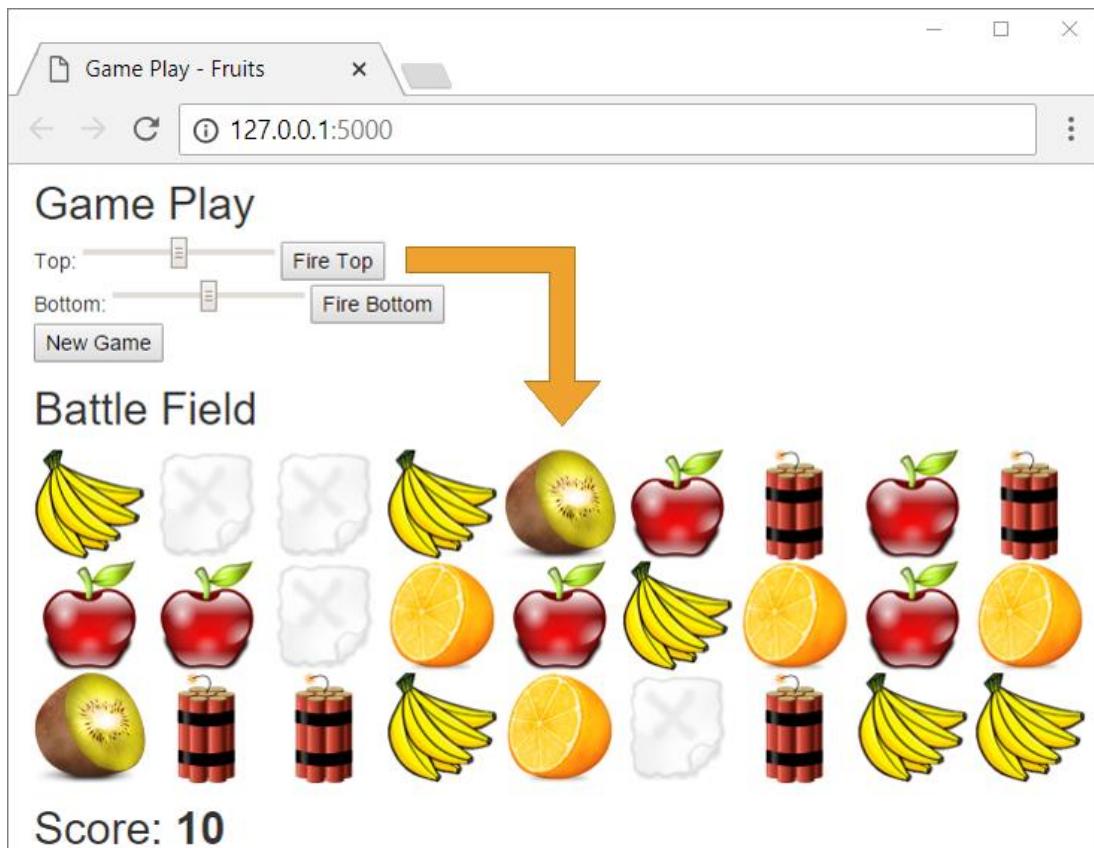
## Упражнения: уеб приложения с по-сложни цикли

Сега вече знаем как да повтаряме група действия, използвайки **цикли**. Нека направим нещо интересно: **уеб базирана игра**. Да, истинска игра, с графика, с гейм логика. Да се позабавляваме. Ще бъде сложно, но ако не разберете нещо как точно работи, няма проблем. Сега още навлизаме в програмирането. Има време, ще напреднете с технологиите. Засега следвайте стъпките.

### Задача: уеб игра "Обстреляй плодовете!"

**Условие:** Да се разработи **Flask** уеб приложение – игра, в която играчът стреля по **плодове**, подредени в таблица. Успешно уцелените плодове изчезват, а играчът получава точки за всеки уцелен плод. При уцелване на **динамит**, плодовете се взривяват и играта свършва (ако в игри като Fruit Ninja). Стрелбата се извършва по колони, отгоре надолу или отдолу нагоре, а местоположението на удара (колоната под обстрел) се задава чрез скролер (scroll bar). Заради неточността на скролера, играчът не е съвсем сигурен по коя колона ще стреля. Така при всеки изстрел има шанс да не улучи и това прави играта по-интересна (подобно на прашката в Angry Birds).

Играча ни трябва да изглежда по този начин:

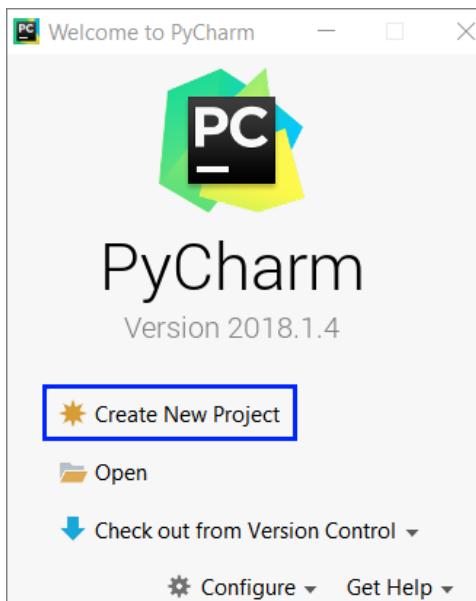




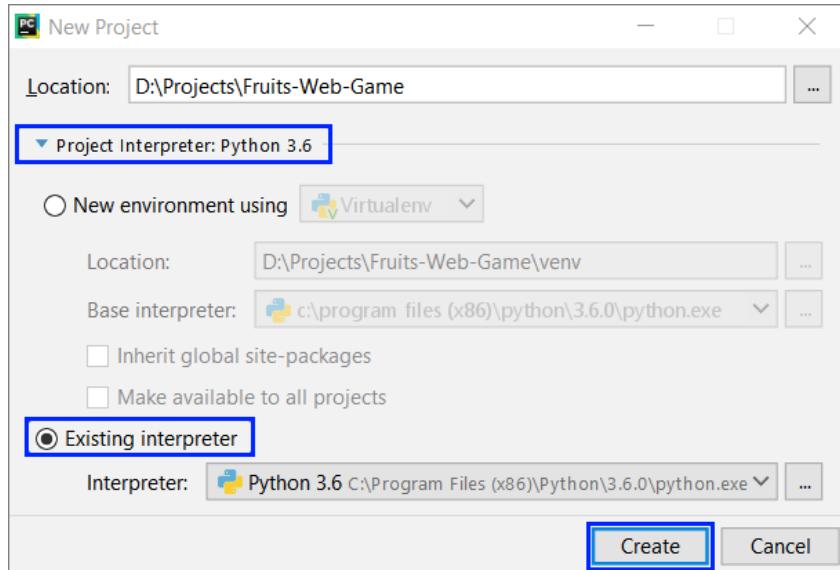
Следват стъпките за Python имплементация на уеб приложението "Обстреляй плодовете!".

## Празно PyCharm решение

Създаваме празно решение в PyCharm, за да си организираме разбираемо кода от приложението:



След това, задаваме съмислено име на проекта, например "Fruits-Web-Game". Също така, задаваме Python интерпретатора на този по подразбиране:

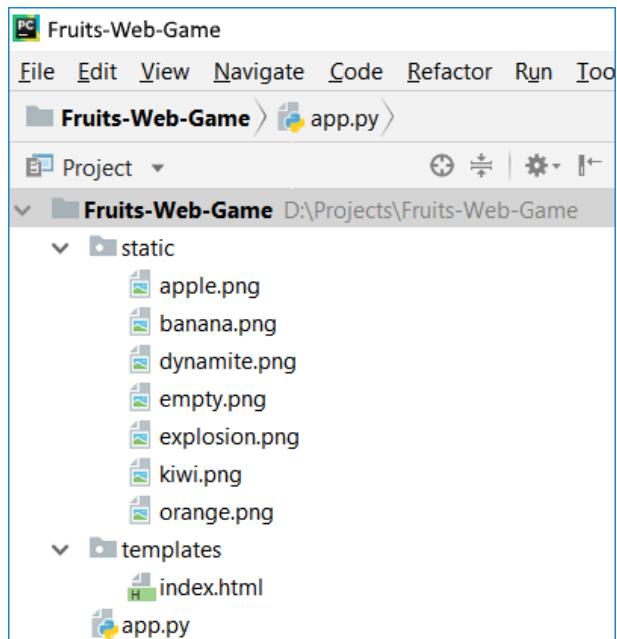


Ще създадем нашето уеб приложение, ползвайки библиотеката **Flask**, с която вече се запознахме. Както вече знаем, преди да започнем да пишем код, трябва да я инсталираме. Отиваме в настройките на PyCharm [File] -> [Settings] и след това в [Project: Fruit-Web-Game] -> [Project Interpreter]. Там, натискаме бутона **+**, търсим и инсталираме **Flask**.

Сега трябва да вземем структурата на проекта от предоставените ни ресурси.

След това, добавяме **ресурсите** за играта (те са част от файловете със заданието за този проект и могат да бъдат свалени от GitHub: <https://github.com/SoftUni/Programming-Basics-Book-Python-BG/tree/master/assets/chapter-7-1-assets>). Копираме ги от Windows Explorer и ги поставяме в папката на проекта в PyCharm с **copy/paste**. След като поставим ресурсите, структурата на проекта трябва да изглежда като на картинката.

Ако отворим **app.py** и го пуснем с десен бутон -> [Run 'app'], би трявало да се отвори приложението, след което да натиснем върху линка в конзолата и да се отвори нашия уеб браузър:



```

from flask \
    import Flask, render_template, redirect, request
import random

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

if __name__ == '__main__':
    app.run()

```

Run: app

```

"C:\Program Files (x86)\Python36\python.exe" "C:/Users/PC/PycharmProjects/Fruits-Web-Game/app.py"
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

Packages installed successfully: Installed packages: 'Flask' (23 minutes ago) 7:35 CRLF UTF-8

Сега създаваме контролите за играта. Целта е да добавим **скролиращи ленти** (scroll bars), с които играчът да се прицелва, и бутон за стартиране на **нова игра**. Затова трябва да редактираме файла **templates/index.html**. Изтриваме "Hello World" и на негово място въвеждаме кода от картинката по-долу.

Този код създава уеб форма **<form>** със скролер (поле) **position** за задаване на число в интервала [0 ... 100] и бутон **[Fire Top]** за изпращане на данните от формата към сървъра. Действието, което ще обработи данните, се казва **/FireTop**, което означава функция **fire\_top()**, която се намира във файла **app.py**. Следват още две подобни форми с бутона **[Fire Bottom]** и **[New Game]**.

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** Fruits-Web-Game
- Menu Bar:** File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help
- Toolbar:** app, Run, Stop, Refresh, Search
- Project Explorer:**
  - Fruits-Web-Game (D:\Project)
  - static:
    - apple.png
    - banana.png
    - dynamite.png
    - empty.png
    - explosion.png
    - kiwi.png
    - orange.png
  - templates:
    - index.html
- Code Editor:** Shows the content of index.html:
 

```

1 <html>
2   <head>
3     <title>Game Play - Fruits</title>
4   </head>
5   <body>
6     <h2>Game Play</h2>
7
8     <form action="/FireTop">
9       Top: <input type="range"
10        name="position" min="0" max="100"/>
11       <input type="submit" value="Fire Top">
12     </form>
13     <form action="/FireBottom">
14       Bottom: <input type="range"
15        name="position" min="0" max="100"/>
16       <input type="submit" value="Fire Bottom"/>
17     </form>
18     <form action="/Reset">
19       <input type="submit" value="New Game"/>
20     </form>
21   </body>
22 </html>
      
```
- Status Bar:** Packages installed successfully: Installed packages: 'Flask' (30 minutes ago) | 5:7 CRLF UTF-8 | 1

Сега трябва да подгответим плодовете за рисуване в изгледа (view). Добавяме кода от картинката по-долу в **app.py** файла.

Този код дефинира полета за **брой редове**, **брой колони**, за **таблицата с плодовете** (игралното поле), за натрупаните от играта **точки** и информация дали играта е активна или е **свършила** (поле **gameOver**). Игралиятото поле е с размери 9 колони на 3 реда и съдържа за всяко поле текст какво има в него: **apple**, **banana**, **orange**, **kiwi**, **empty** или **dynamite**. Главното действие **index()** подгответя игралното поле за чертане като записва елементите на играта и извиква изгледа, който ги чертае в страницата на играта (в уеб браузъра като HTML).

Трябва да генерираме случаини плодове. За да направим това, трябва да напишем метод **generate\_random\_fruits()** с кода от картинката по-долу. Този код записва в таблицата (матрицата) **fruits** имена на различни картинки и така изгражда игралното поле. Във всяка клетка от таблицата се записва една от следните стойности: **apple**, **banana**, **orange**, **kiwi**, **empty** или **dynamite**. След това, за да се нарисува съответното изображение в изгледа, към текста от таблицата ще се долепи **.png** и така ще се получи името на файла с картинката, която да се

вмъкне в HTML страницата като част от игралното поле. Попълването на игралното поле (9 колони с по 3 реда) става в изгледа **index.html**, с два вложени **for** цикъла (за ред и за колона).

```

1  from flask \
2      import Flask, render_template, redirect, request
3      import random
4
5      app = Flask(__name__)
6
7      rows = 3
8      cols = 9
9
10     fruits = generate_random_fruits()
11
12     score = 0
13     game_over = False
14
15     @app.route('/')
16     def index():
17         return render_template('index.html',
18             rows=rows, cols=cols, fruits=fruits,
19             game_over=game_over, score=score)
20
21     if __name__ == '__main__':
22         app.run()

```

Packages installed successfully: Installed packages: 'Flask' 21:14 CRLF UTF-8 🏠 🎨 1

За да се генерират случайни плодове, за всяка клетка се генерира **случайно** число между 0 и 8 (вж. класа **random** в Python). Ако числото е 0 или 1, се слага **apple**, ако е между 2 и 3, се слага **banana** и т.н. Ако числото е 8, се поставя **dynamite**. Така плодовете се появяват 2 пъти по-често отколкото динамита.

Кодът с двата вложени **for** цикъла за генериране на случайни плодове трябва да се сложи във файла **app.py**. Той е даден по-долу.

The screenshot shows the PyCharm IDE interface with the following details:

- Project Structure:** The left sidebar shows the project structure under "Fruits-Web-Game". It includes a "static" folder containing various PNG files (apple, banana, dynamite, empty, explosion, kiwi, orange) and a "templates" folder containing "index.html" and "app.py".
- Code Editor:** The main window displays the content of "app.py". The code defines a function "generate\_random\_fruits" which generates a 3x9 grid of random fruits. The logic uses nested loops and conditional statements to place "apple", "banana", "orange", "kiwi", or "dynamite" at each position. A return statement returns the list of lists representing the grid.
- Terminal:** At the bottom, a terminal window shows the message: "Packages installed successfully: Installed packages: 'Flask' (49 minutes ago)".

```

5     rows = 3
6
7     cols = 9
8
9     def generate_random_fruits():
10        fruits = [[] for row in range(rows)]
11
12        for row in range(rows):
13            for col in range(cols):
14                r = random.randint(0, 8)
15                if r < 2:
16                    fruits[row].append('apple')
17                elif r < 4:
18                    fruits[row].append('banana')
19                elif r < 6:
20                    fruits[row].append('orange')
21                elif r < 8:
22                    fruits[row].append('kiwi')
23                else:
24                    fruits[row].append('dynamite')
25
26        return fruits
27
28
29    fruits = generate_random_fruits()
30
31    score = 0
32    game_over = False
33
34    generate_random_fruits()

```

За да попълним игралното поле с плодовете, трябва да завъртим **два вложени цикъла** (за редовете и за колоните). Всеки ред се състои от 9 на брой картички, всяка от които съдържа **apple**, **banana** или друг плод, или празно поле **empty**, или **dynamite**. Картинките се чертаят като се отпечатат HTML таг за вмъкване на картинка: ``. Девет картички се подреждат една след друга на всеки от редовете, а след тях се преминава на нов ред с `<br>`. Това се повтаря три пъти за трите реда. Накрая се отпечатват точките на играча.

Обърнете внимание на къдравите скоби – те служат за превключване между езика **HTML** и езика **Python** и идват от **Jinja2** синтаксиса за рисуване на динамични уеб страници.

Ето как изглежда **кодът** за чертане на игралното поле и точките в **index.html**:

PC Fruits-Web-Game

File Edit View Navigate Code Refactor Run Tools VCS Window Help

Fruits-Web-Game > templates > index.html

Project D:\Proj...

app.py index.html

```

11   </form>
12
13   <form action="/FireBottom">
14     Bottom: <input type="range" name="position"
15       min="0" max="100"/>
16     <input type="submit" value="Fire Bottom"/>
17   </form>
18
19   <form action="/Reset">
20     <input type="submit" value="New Game"/>
21   </form>
22
23   <h2>Battle Field</h2>
24   {% for row in range(rows) %}
25     {% for col in range(cols) %}
26       
27     {% endfor %}
28     <br />
29   {% endfor %}
30   <h2>Score: <b>{{score}}</b></h2>
31

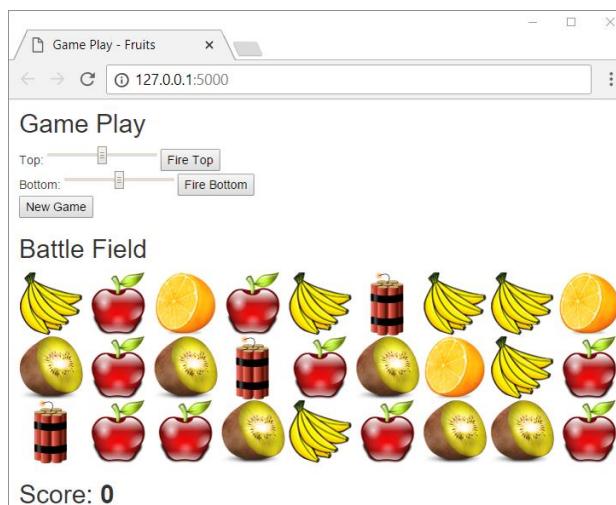
```

External Libraries

Scratches and Consoles

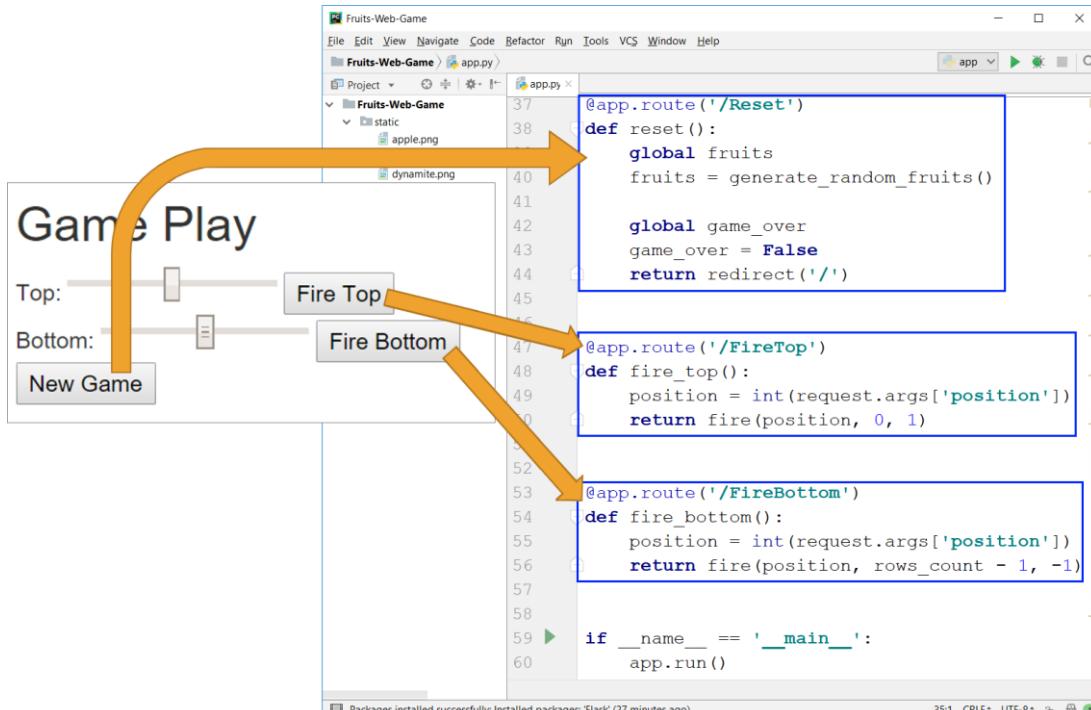
Packages installed successfully: Installed packages: 'Flask' (54 minutes ago) 22:22 CRLF UTF-8

Стартираме проекта с [Shift + F10]. Очаква се да бъде генерирано случаен игрово поле с плодове с размери 9 на 3 и да се визуализира в уеб страницата чрез поредица картинки:



Сега играта е донякъде направена: игралното поле се генерира случайно и се визуализира успешно (ако не сте допуснали грешка някъде). Остава да реализираме същината на играта: стрелянето по плодовете.

За целта добавяме действията [New Game], [Fire Top] и [Fire Bottom] във файла `app.py`:



Чрез горния код дефинираме три действия:

- **reset()** – стартира нова игра, като генерира ново случайно игрално поле с плодове и експлозиви, нулира точките на играча и прави играта валидна (`gameOver = False`). Това действие е доста просто и може да се тества веднага с `[Shift+F10]`, преди да се напишат другите.
- **fire\_top()** – стреля по ред 0 на позиция `position` (число от 0 до 100), взета от потребителя. Извиква се стреляне в посока **надолу** (+1) от ред 0 (най-горния). Самото стреляне е по-сложно като логика и ще бъде разгледано след малко.
- **fire\_bottom()** – стреля по ред 2 на позиция `position` (число от 0 до 100), взета от потребителя. Извиква се стреляне в посока **нагоре** (-1) от ред 2 (най-долния).

Имплементираме "стрелянето" – метода `fire(position, start_row, step)`:

```

def fire(position, start_row, step):
    col = position * (cols - 1) // 100
    row = start_row
    while row >= 0 and row < rows:
        fruit = fruits[row][col]
        if fruit in ['apple', 'banana', 'orange', 'kiwi']:
            global score
            score += 1
            fruits[row][col] = 'empty'
            break
        elif fruit == 'dynamite':
            global game_over
            game_over = True
            break
        row = row + step
    return redirect('/')

@app.route('/FireTop')
def fire_top():
    position = int(request.args['position'])
    return fire(position, 0, 1)

@app.route('/FireBottom')
def fire_bottom():
    position = int(request.args['position'])
    return fire(position, rows - 1, -1)

```

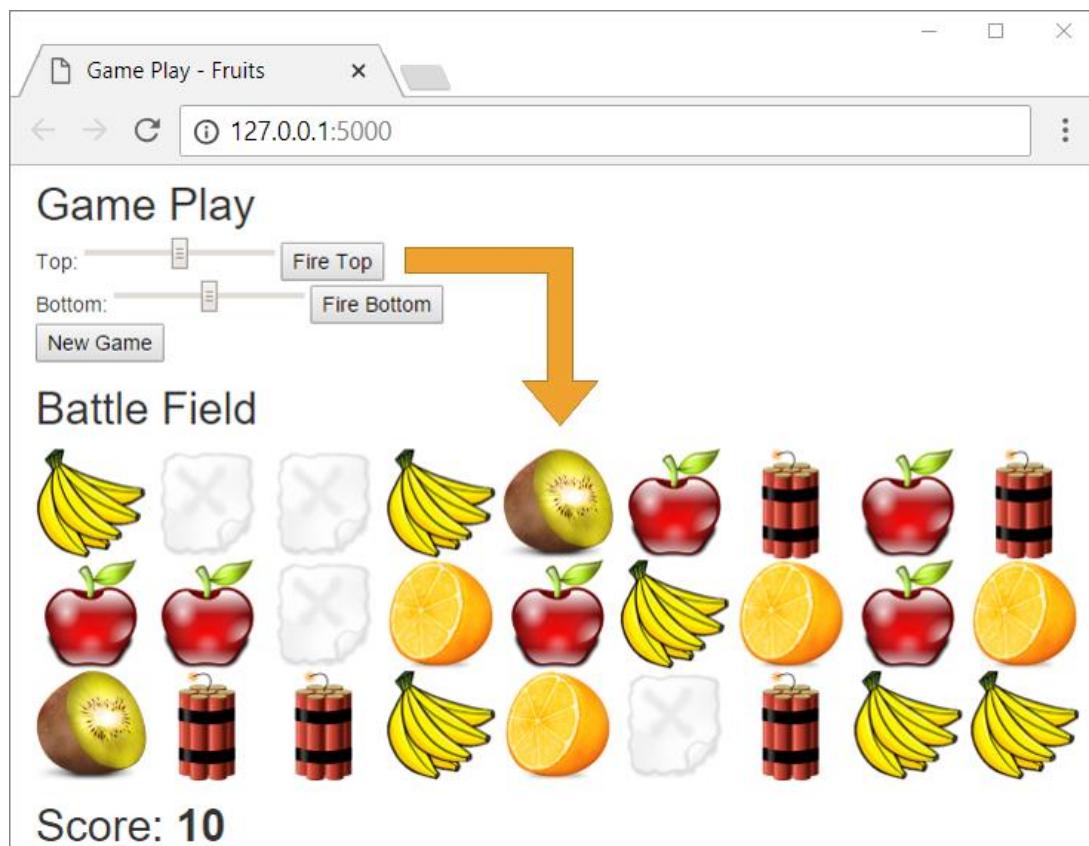
Стрелянето работи по следния начин: първо се изчислява номера на колоната **col**, към която играчът се е прицелил. Входното число от скролера (между 0 и 100) се намаля до число между 0 и 8 (за всяка от 9-те колони). Номерът на реда **row** е или 0 (ако изстрелът е отгоре) или броят редове минус едно (ако изстрелът е отдолу). Съответно посоката на стрелба (стъпката) е **1** (надолу) или **-1** (нагоре).

За да се намери къде изстрелът поразява плод или динамит, се преминава в цикъл през всички клетки от игралното поле в прицелената колона и от първия до последния атакуван ред. Ако се срещне плод, той изчезва (замества се с **empty**) и се дават точки на играча. Ако се срещне **dynamite**, играта се отбелязва като свършила.

Оставаме на по-запалените читатели да имплементират по-сложно поведение, например да се дават различни точки при уцелване на различен плод, да се реализира анимация с експлозия (това не е твърде лесно), да се взимат точки при излишно стреляне в празна колона и подобни.

Тестваме какво работи до момента като стартираме приложението с [Ctrl + Shift + F10]:

- **Нова игра** → бутона за нова игра трябва да генерира ново игрално поле със случаино разположени плодове и експлозиви и да нулира точките на играча.
- **Стреляне отгоре** → стрелянето отгоре трябва да премахва най-горния плод в уцелената колона или да предизвиква край на играта при динамит. Въсъщност при край на играта все още нищо няма да се случва, защото в изгледа този случай още не се разглежда.
- **Стреляне отдолу** → стрелянето отдолу трябва да премахва най-долния плод в уцелената колона или да прекратява играта при уцелване на динамит.



За момента при "Край на играта" нищо не се случва. Ако играчът уцели динамит, в приложението се отбележва, че играта е свършила (**game\_over = True**), но този факт не се визуализира по никакъв начин. За да заработи приключването на играта, е необходимо да добавим няколко проверки в изгледа.

Кодът по-долу проверява дали е свършила играта и показва съответно контролите за стреляне и игралното поле (при активна игра) или картинка с експлодирали плодове при край на играта.

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** Fruits-Web-Game
- Menu Bar:** File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help
- Toolbar:** app, Run, Stop, Refresh, Search
- Project Explorer:** Fruits-Web-Game (selected), static (contains apple.png, banana.png, dynamite.png, empty.png, explosion.png, kiwi.png, orange.png), templates (contains index.html, app.py), External Libraries, Scratches and Console.
- Code Editor:** The file index.html is open. The code contains logic for a game where the user can fire from the top or bottom. It checks if the game is over and displays an explosion image if it is. It also displays the current score. The code is as follows:

```

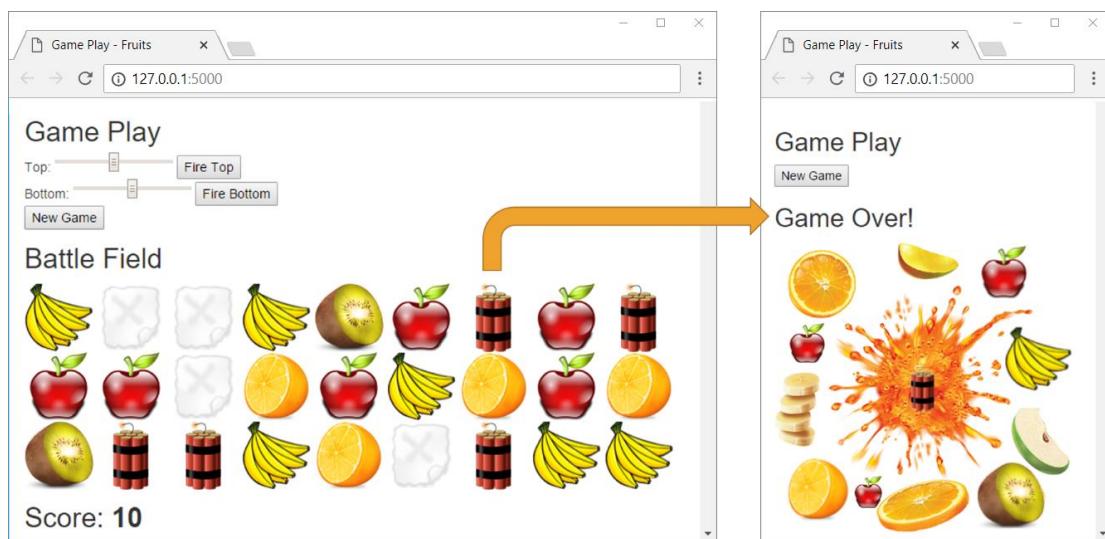
<h2>Game Play</h2>
{% if not game_over %}
    <form action="/FireTop">
        Top: <input type="range" name="position" min="0" max="100"/>
        <input type="submit" value="Fire Top">
    </form>
    <form action="/FireBottom">
        Bottom: <input type="range" name="position" min="0" max="100"/>
        <input type="submit" value="Fire Bottom"/>
    </form>
{% endif %}
<form action="/Reset">
    <input type="submit" value="New Game"/>
</form>
{% if game_over %}
    <h2>Game Over!</h2>
    
{% else %}
    <h2>Battle Field</h2>
    {% for row in range(rows) %}
        {% for col in range(cols) %}
            
        {% endfor %}
        <br />
    {% endfor %}
    <h2>Score: <b>{{score}}</b></h2>
{% endif %}
</body>
</html>

```
- Bottom Status Bar:** Packages installed successfully: Installed packages: 'Flask' (today 19:40) 7:23 CRLF UTF-8

След промяната в кода на изгледа стартираме с [Ctrl + Shift + F10] и тестваме играта отново.

Този път при уцелване на динамит, трябва да се появи дясната картичка и да се позволява единствено действието "нова игра" (бутона [New Game]).

Ето как изглежда играта в действие, когато е готова:



Сложно ли беше? Успяхте ли да направите играта? Ако не сте успели, не се притеснявайте, това е сравнително сложен проект, който включва голяма доза не изучавана материя. Ако срещнете някакви затруднения, може да питате във форума на СофтУни: <https://softuni.bg/forum>.

# Глава 7.2. По-сложни цикли – изпитни задачи

Вече научихме как може да изпълним даден блок от команди повече от веднъж използвайки **for** цикъл. В предходната глава разгледахме още няколко конструкции за цикъл, които биха ни помогнали при решаването на по-сложни проблеми, а именно:

- цикли със стъпка
- вложени цикли
- **while** цикли
- **while + break** цикли
- безкрайни цикли и излизане от цикъл (с **break** оператора)
- конструкцията **try-except**

## Изпитни задачи

В тази глава ще затвърдим знанията си като решим няколко по-сложни задачи с цикли, давани на приемни изпити.

### Задача: генератор за тъпи пароли

Да се напише програма, която въвежда две цели числа **n** и **l** и генерира по азбучен ред всички възможни "тъпи" пароли", които се състоят от следните 5 символа:

- Символ 1: цифра от 1 до n.
- Символ 2: цифра от 1 до n.
- Символ 3: малка буква измежду първите l букви на латинската азбука.
- Символ 4: малка буква измежду първите l букви на латинската азбука.
- Символ 5: цифра от 1 до n, по-голяма от първите 2 цифри.

### Примерен вход и изход

Вход	Изход	Вход	Изход
2	11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2 11ca2 11cb2 11cc2 11cd2	3	11aa2 11aa3 12aa3
4	11da2 11db2 11dc2 11dd2	1	21aa3 22aa3

Вход	Изход	Вход	Изход
4 2	11aa2 11aa3 11aa4 11ab2 11ab3 11ab4 11ba2 11ba3 11ba4 11bb2 11bb3 11bb4 12aa3 12aa4 12ab3 12ab4 12ba3 12ba4 12bb3 12bb4 13aa4 13ab4 13ba4 13bb4 21aa3 21aa4 21ab3 21ab4 21ba3 21ba4 21bb3 21bb4 22aa3 22aa4 22ab3 22ab4 22ba3 22ba4 22bb3 22bb4 23aa4 23ab4 23ba4 23bb4 31aa4 31ab4 31ba4 31bb4 32aa4 32ab4 32ba4 32bb4 33aa4 33ab4 33ba4 33bb4	3 2	11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3 12aa3 12ab3 12ba3 12bb3 21aa3 21ab3 21ba3 21bb3 22aa3 22ab3 22ba3 22bb3

## Входни данни

Входът се чете от конзолата и се състои от **две цели числа: n и l** в интервала [1 ... 9], по едно на ред.

## Изходни данни

На конзолата трябва да се отпечатат **всички "тъпи"** пароли по азбучен ред, разделени с **интервал**.

## Насоки и подсказки

Решението на задачата можем да разделим мислено на три части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на две числа **n** и **l**, всяко на отделен ред.
- **Обработка на входните данни** – използване на вложени цикли за преминаване през всеки възможен символ, за всеки от петте символа на паролата.
- **Извеждане на резултат** – отпечатване на всяка "тъпа" парола, която отговаря на условията.

## Прочитане и обработка на входните данни

За прочитане на **входните** данни ще декларираме две променливи от целочислен тип **int: n** и **l**.

```
n = int(input())
l = int(input())
```

## Извеждане на резултат

Един от начините да намерим решението на тази задача е да създадем **пет for** цикъла, вложени един в друг, по един за всеки символ. За да гарантираме

условието последния символ, който по условие е число, да бъде **по-голям** от първите два, които също са числа, ще използваме вградената функция **max(...)**:

```
result = ''
for s1 in range(1, n + 1):
    for s2 in range(1, n + 1):
        for s3 in range(97, 1 + 97):
            for s4 in range(97, 1 + 97):
                for s5 in range(max(s1,s2) + 1, n + 1):
                    result += ('' + str(s1) + str(s2)
                               + chr(s3) + chr(s4)
                               + str(s5) + ' ')
print(result.strip())
```

Знаете ли, че...?

- За удобство можем да използваме директно ASCII стойността на символа "a" в декларациите на циклите - **97**:

```
for s3 in range(97, 1 + 97):
    for s4 in range(97, 1 + 97):
```

- Ако не сме сигурни в дадена стойност на символ или искаме да взимаме стойностите на динамично генериирани символи, можем да използваме вградената функция **ord(char)**, която да ни върне стойността на символа:

```
for s3 in range(ord('a'), 1 + ord('a')):
    for s4 in range(ord('a'), 1 + ord('a')):
```

- Или обратното, можем да генерираме символ спрямо зададена стойност, използвайки вградена функция в Python **chr(number)**:

```
result += ('' + str(s1) + str(s2)
           + chr(s3) + chr(s4)
           + str(s5) + ' ')
```

- Python разполага с богат набор от вградени функции и методи за работа със стрингове. Потърсете допълнителна информация за следните няколко текстови функции: **str(number)**, **.lower()**, **.format(\*args, \*\*kwargs)**, **.swapcase()**, **.upper()**.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1058#0>.

## Задача: магически числа

Да се напише програма, която въвежда едно цяло **магическо** число и изкарва всички възможни **6-цифрени** числа, за които произведението на техните цифри е равно на магическото число.

Пример: "Магическо число" → 2

- 111112 →  $1 * 1 * 1 * 1 * 1 * 2 = 2$
- 111121 →  $1 * 1 * 1 * 1 * 2 * 1 = 2$
- 111211 →  $1 * 1 * 1 * 2 * 1 * 1 = 2$
- 112111 →  $1 * 1 * 2 * 1 * 1 * 1 = 2$
- 121111 →  $1 * 2 * 1 * 1 * 1 * 1 = 2$
- 211111 →  $2 * 1 * 1 * 1 * 1 * 1 = 2$

## Входни данни

Входът се чете от конзолата и се състои от **цяло число** в интервала [1 ... 600 000].

## Изходни данни

На конзолата трябва да се отпечатат **всички магически числа**, разделени с **интервал**.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2	111112 111121 111211 112111 121111 211111	8	111118 111124 111142 111181 111214 111222 111241 111412 111421 111811 112114 112122 112141 112212 112221 112411 114112 114121 114211 118111 121114 121122 121141 121212 121221 121411 122112 122121 122211 124111 141112 141121 141211 142111 181111 211114 211122 211141 211212 211221 211411 212112 212121 212211 214111 221112 221121 221211 222111 241111 411112 411121 411211 412111 421111 811111	53144 1	999999

## Насоки и подсказки

Решението на задачата за магическите числа следва **същата** концепция (отново трябва да генерираме всички комбинации за п елемента). Следвайки стъпките по-долу, се опитайте да решите задачата сами.

- Декларирайте и инициализирайте **променлива** от целочислен тип **int** и прочетете **входа** от конзолата.

- Вложете **шест for цикъла** един в друг, по един за всяка цифра на търсените 6-цифрени числа.
- В последния цикъл, чрез **if** конструкция проверете дали **произведението** на шестте цифри е **равно** на **магическото** число.

Ето примерна имплементация на идеята:

```
n = 123456
result = ''
for a in range(1, 10):
    for b in range(1, 10):
        for c in range(1, 10):
            for d in range(1, 10):
                for e in range(1, 10):
                    for f in range(1, 10):
                        if a * b * c * d * e * f == n:
                            result += str(a) + str(b) + str(c) + str(d) + str(e) + str(f)

print(result)
```

В предходната глава разглеждахме и други циклични конструкции. Нека разгледаме примерно решение на същата задача, в което използваме цикъла **while**. Първо трябва да запищем **входното магическо число** в подходяща променлива:

```
n = int(input())
result = ''
a = 1
```

След това ще започнем да разписваме **while** циклите.

- Ще инициализираме **първата цифра**: **a = 1**.
- Ще зададем **условие за всеки цикъл**: цифрата да бъде по-малка или равна на 9.
- В началото на всеки цикъл задаваме стойност на **следващата цифра**, в случая: **b = 1**. При вложените **for** цикли инициализираме променливите във вътрешните цикли при всяко увеличение на външните. Искаме да постигнем същото поведение и тук.
- В **края** на всеки цикъл ще **увеличаваме** съответната цифра с едно: **a += 1, b += 1, c += 1** и т.н.
- В **най-вътрешния** цикъл ще направим **проверката** и ако е необходимо, ще принтираме на конзолата.

```

while a <= 9:
    b = 1
    while b <= 9:
        c = 1
        while c <= 9:
            d = 1
            while d <= 9:
                e = 1
                while e <= 9:
                    f = 1
                    while f <= 9:
                        if a * b * c * d * e * f == n:
                            result += ('' + str(a) + str(b)
                                       + str(c) + str(d)
                                       + str(e) + str(f) + ' ')
                    f += 1
                e += 1
            d += 1
        c += 1
    b += 1
a += 1
print(result)

```

Както виждаме, един проблем може да бъде решен с различни видове цикли. Разбира се, за всяка задача има най-подходящ избор. С цел да упражните всеки цикъл - опитайте се да решите всяка от следващите задачи с всички изучени цикли.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1058#1>.

## Задача: спиращо число

Напишете програма, която принтира на конзолата всички числа от **N** до **M**, които се делят на 2 и на 3 без остатък, в обратен ред. От конзолата ще се чете още **едно** "спиращо" число **S**. Ако някое от делящите се на 2 и 3 числа е **равно на спиращото число**, то не трябва да се принтира и програмата трябва да приключи. В противен случай се принтират всички числа до **N**, които отговарят на условието.

### Вход

От конзолата се четат 3 числа, всяко на отделен ред:

- **N** - цяло число:  $0 \leq N < M$ .

- $M$  - цяло число:  $N < M \leq 10000$ .
- $S$  - цяло число:  $N \leq S \leq M$ .

## Изход

На конзолата се принтират на един ред, разделени с интервал, всички числа, отговарящи на условията.

## Примерен вход и изход

Вход	Изход	Обяснения
1 30 15	30 24 18 12 6	Числата от 30 до 1, които се делят едновременно на 2 и на 3 без остатък са: 30, 24, 18, 12 и 6, а 15 не е равно на нито едно, затова редицата продължава.

Вход	Изход	Обяснения
1 36 12	36 30 24 18	Числата от 36 до 1, които се делят едновременно на 2 и на 3 без остатък, са: 36, 30, 24, 18, 12 и 6. Числото 12 е равно на спиращото число, затова спираме до 18.

## Насоки и подсказки

Задачата може да се раздели на **четири** логически части:

- Прочитане на входните данни от конзолата.
- Проверка на всички числа в дадения интервал (съответно чрез завъртане на цикъл).
- Проверка на условията от задачата спрямо всяко едно число от въпросния интервал.
- Отпечатване на числата.

Първата част е тривиална - прочитаме **три** цели числа от конзолата, съответно ще използваме тип **int**.

С **втората** част също сме се сблъсквали - инициализиране на **for** цикъл. Тук има малка **уловка** - в условието е споменато, че числата трябва да се принтират в **обратен ред**. Това означава, че **началната** стойност на променливата **i** ще е **поголямото** число, което от примерите виждаме, че е **M**. Съответно, **крайната** стойност на **i** трябва да е **N**. Фактът, че ще печатаме резултатите в обратен ред и стойностите на **i** ни подсказват, че стъпката ще е **намаляване с 1**:

```
for i in range(m, n - 1, -1):
```

След като сме инициализирали **for** цикъла, идва ред на **третата** част от задачата – **проверка** на условието дали даденото **число се дели на 2 и на 3 без остатък**. Това ще направим с една обикновена **if** проверка, която ще оставим на читателя сам да състави.

Другата **уловка** в тази задача е, че освен горната проверка, трябва да направим **още** една – дали **числото е равно на "спиращото" число**, подадено ни от конзолата на третия ред. За да се стигне до тази проверка, числото, което проверяваме, трябва да премине през горната. По тази причина ще построим **още** една **if** конструкция, която ще **вложим в предходната**. Ако условието е **вярно**, заданието е да спрем програмата да печата, което в конкретния случай можем да направим с оператор **break**, който ще ни **изведе** от **for** цикъла.

Съответно, ако **условието** на проверката дали числото съвпада със "спиращото" число върне резултат **false**, по задание нашата програма трябва да **продължи да печата**. Това въщност покрива и **четвъртата и последна** част от нашата програма.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1058#2>.

## Задача: специални числа

Да се напише програма, която **въвежда** едно **цяло** **число N** и генерира всички възможни **"специални"** **числа** от **1111** до **9999**. За да бъде **"специално"** едно число, то трябва да отговаря на **следното условие**:

- N да се дели на всяка една от неговите цифри без остатък.

**Пример:** при **N = 16, 2418** е специално число:

- $16 / 2 = 8$  без остатък
- $16 / 4 = 4$  без остатък
- $16 / 1 = 16$  без остатък
- $16 / 8 = 2$  без остатък

## Примерен вход и изход

Вход	Изход	Коментари
3	1111 1113 1131 1133 1311 1313 1331 1333 3111 3113 3131 3133 3311 3313 3331 3333	$3 / 1 = 3$ без остатък $3 / 3 = 1$ без остатък $3 / 3 = 1$ без остатък $3 / 3 = 1$ без остатък

Вход	11
Изход	1111
Вход	16
Изход	1111 1112 1114 1118 1121 1122 1124 1128 1141 1142 1144 1148 1181 1182 1184 1188 1211 1212 1214 1218 1221 1222 1224 1228 1241 1242 1244 1248 1281 1282 1284 1288 1411 1412 1414 1418 1421 1422 1424 1428 1441 1442 1444 1448 1481 1482 1484 1488 1811 1812 1814 1818 1821 1822 1824 1828 1841 1842 1844 1848 1881 1882 1884 1888 2111 2112 2114 2118 2121 2122 2124 2128 2141 2142 2144 2148 2181 2182 2184 2188 2211 2212 2214 2218 2221 2222 2224 2228 2241 2242 2244 2248 2281 2282 2284 2288 2411 2412 2414 2418 2421 2422 2424 2428 2441 2442 2444 2448 2481 2482 2484 2488 2811 2812 2814 2818 2821 2822 2824 2828 2841 2842 2844 2848 2881 2882 2884 2888 4111 4112 4114 4118 4121 4122 4124 4128 4141 4142 4144 4148 4181 4182 4184 4188 4211 4212 4214 4218 4221 4222 4224 4228 4241 4242 4244 4248 4281 4282 4284 4288 4411 4412 4414 4418 4421 4422 4424 4428 4441 4442 4444 4448 4481 4482 4484 4488 4811 4812 4814 4818 4821 4822 4824 4828 4841 4842 4844 4848 4881 4882 4884 4888 8111 8112 8114 8118 8121 8122 8124 8128 8141 8142 8144 8148 8181 8182 8184 8188 8211 8212 8214 8218 8221 8222 8224 8228 8241 8242 8244 8248 8281 8282 8284 8288 8411 8412 8414 8418 8421 8422 8424 8428 8441 8442 8444 8448 8481 8482 8484 8488 8811 8812 8814 8818 8821 8822 8824 8828 8841 8842 8844 8848 8881 8882 8884 8888

## Входни данни

Входът се чете от конзолата и се състои от **цяло число** в интервала [1 ... 600 000].

## Изходни данни

Да се отпечатат на конзолата **всички специални числа**, разделени с **интервал**.

## Насоки и подсказки

Решете задачата самостоятелно използвайки наученото от предишните две. Спомнете си разликата между операторите за **целочислено деление /** и **деление с остатък %** в Python.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1058#3>.

## Задача: цифри

Да се напише програма, която прочита от конзолата 1 цяло число в интервала [100 ... 999], и след това го принтира определен брой пъти - модифицирайки го преди всяко принтиране по следния начин:

- Ако числото се дели на 5 без остатък, **извадете** от него **първата** **му** **цифра**.
- Ако числото се дели на 3 без остатък, **извадете** от него **втората** **му** **цифра**.
- Ако нито едно от горните условия не е вярно, **прибавете** към него **третата** **му** **цифра**.

Принтирайте на конзолата **N** **брой реда**, като всеки ред има **M** **на** **брой** **числа**, които са резултат от горните действия. Нека:

- **N** = **сбора** на **първата** и **втората** **цифра** на **числото**.
- **M** = **сбора** на **първата** и **третата** **цифра** на **числото**.

## Входни данни

Входът се чете от конзолата и е цяло число в интервала [100 ... 999].

## Изходни данни

На конзолата трябва да се отпечатат **всички цели числа**, които са резултат от дадените по-горе изчисления в съответния брой редове и колони, както в примерите.

## Примерен вход и изход

Вход	Изход	Коментари
376	382 388 394 400 397 403 409 415 412 418 424 430 427 433 439 445 442 448 454 460 457 463 469 475 472 478 484 490 487 493 499 505 502 508 514 520 517 523 529 535 532 538 544 550 547 553 559 565 562 568 574 580 577 583 589 595 592 598 604 610 607 613 619 625 622 628 634 640 637 643 649 655 652 658 664 670 667 673 679 685 682 688 694 700 697 703 709 715 712 718	10 реда по 9 числа на всеки. Входното число 376: 376 → нито на 5, нито на 3 → 376 + 6 → = = 382 → нито на 5, нито на 3 → 382 + 6 = = 388 + 6 = 394 + 6 = 400 → деление на 5 → 400 - 3 = 397

Вход	Изход	Коментари
132	129 126 123 120 119 121	$(1 + 3) = 4$ и $(1 + 2) = 3 \rightarrow$ 4 реда по 3 числа на всеки.

Вход	Изход	Коментари
	123 120 119 121 123 120	Входното число 132: 132 → деление на 3 → $132 - 3 =$ $= 129 \rightarrow$ деление на 3 → $129 - 3 =$ $= 126 \rightarrow$ деление на 3 → $126 - 3 =$ $= 123 \rightarrow$ деление на 3 → $123 - 3 =$ $= 120 \rightarrow$ деление на 5 → $120 - 1 =$ ..... $121 \rightarrow$ нито на 5, нито на 3 → $121 + 2 = 123$

## Насоки и подсказки

Решете задачата **самостоятелно**, използвайки наученото от предходните. Не забравяйте, че ще е нужно да дефинирате **отделна** променлива за всяка цифра на входното число.

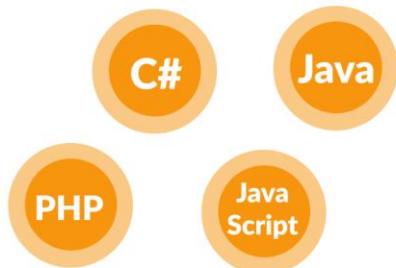
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1058#4>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**СофтУни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофтУни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**СофтУни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 8.1. Подготовка за практически изпит – част I

В настоящата глава ще разгледаме няколко задачи с ниво на трудност, каквото може да очаквате от задачите на практическия изпит по “Основи на програмирането”. Ще преговорим и упражним всички знания, които сме придобили от настоящата книга и през курса "Programming Basics".

## Видео

Гледайте видео-урок по тази глава: [https://youtube.com/watch?v=OYuT5\\_j1OVE](https://youtube.com/watch?v=OYuT5_j1OVE).

## Практически изпит по “Основи на програмирането”

Курсът "Programming Basics" приключва с практически изпит. Включени са 6 задачи, като ще имате 4 часа, за да ги решите. Всяка от задачите на изпита ще засяга една от изучаваните теми по време на курса. Темите на задачите са както следва:

- Задача с прости сметки (без проверки)
- Задача с единична проверка
- Задача с по-сложни проверки
- Задача с единичен цикъл
- Задача с вложени цикли (чертане на фигурука на конзолата)
- Задача с вложени цикли и по-сложна логика

## Система за онлайн оценяване (Judge)

Всички изпити и домашни се тестват автоматизирано през онлайн Judge система: <https://judge.softuni.bg>. За всяка от задачите има открити (нулеви) тестове, които ще ви помогнат да разберете какво се очаква от задачата и да поправите грешките си, както и състезателни тестове, които са скрити и проверяват дали задачата ви работи правилно. В Judge системата се влиза с вашия softuni.bg акаунт.

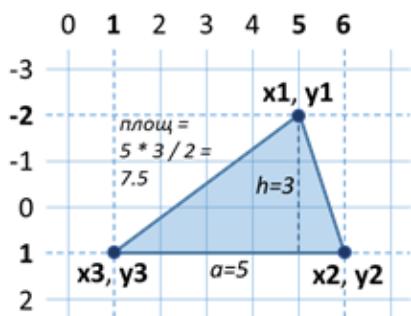
Как работи тестването в Judge системата? Качвате сорс кода и от менюто под него избирате да се изпълни като Python програма. Програмата бива тествана с поредица от тестове, като за всеки успешен тест получавате точки.

## Задачи с прости пресмятания

Първата задача на практическия изпит по “Основи на програмирането” обхваща прости пресмятания без проверки и цикли. Ето няколко примера:

### Задача: лице на триъгълник в равнината

**Триъгълник в равнината** е зададен чрез координатите на трите си върха. Първо е зададен **върхът** ( $x_1, y_1$ ). След това са зададени останалите два върха: ( $x_2, y_2$ ) и ( $x_3, y_3$ ), които **лежат на обща хоризонтална прива** (т.е. имат еднакви Y координати). Напишете програма, която пресмята **лицето на триъгълника** по координатите на трите му върха.



## Вход

От конзолата се четат **6 цели числа** (по едно на ред):  $x_1, y_1, x_2, y_2, x_3, y_3$ .

- Всички входни числа са в диапазона [-1000 ... 1000].
- Гарантирано е, че  $y_2 = y_3$ .

## Изход

Да се отпечата на конзолата **лицето на триъгълника**.

### Примерен вход и изход

Вход	Изход	Чертеж	Обяснения
5 -2 6 1 1 1	7.5		<p>Страната на триъгълника:  <math>a = 6 - 1 = 5</math></p> <p>Височината на триъгълника:  <math>h = 1 - (-2) = 3</math></p> <p>Лицето на триъгълника:  <math>S = a * h / 2 = 5 * 3 / 2 = 7.5</math></p>

Вход	Изход	Чертеж	Обяснения
4 1 -1 -3 3 -3	8		<p>Страната на триъгълника:  <math>a = 3 - (-1) = 4</math></p> <p>Височината на триъгълника:  <math>h = 1 - (-3) = 4</math></p> <p>Лицето на триъгълника:  <math>S = a * h / 2 = 4 * 4 / 2 = 8</math></p>

## Насоки и подсказки

Изключително важно при подобен тип задачи, при които се подават някакви координати, е да обърнем внимание на **реда**, в който се подават, както и правилно да осмислим кои от координатите ще използваме и по какъв начин. В случая, на входа се подават **x1, y1, x2, y2, x3, y3** в този си ред. Ако не спазваме тази последователност, решението става грешно. Първо пишем кода, който чете подадените данни:

```
x1 = int(input())
y1 = int(input())
x2 = int(input())
y2 = int(input())
x3 = int(input())
y3 = int(input())
```

Трябва да пресметнем **страницата** и **височината** на триъгълника. От картинките, както и от условието **y2 = y3** забелязваме, че едната **страница** винаги е успоредна на хоризонталната ос. Това означава, че нейната **дължина** е равна на дълчината на отсечката между нейните координати **x2** и **x3**, която е равна на разликата между по-голямата и по-малката координата. Аналогично можем да изчислим и **височината**. Тя винаги ще е равна на разликата между **y1** и **y2** (или **y3**, тъй като са равни). Тъй като не знаем дали винаги **x2** ще е по-голям от **x3**, или **y1** ще е под или над страницата на триъгълника, ще използваме **абсолютните стойности** на разликата, за да получаваме винаги положителни числа, понеже една отсечка не може да има отрицателна дължина:

```
a = abs(x2 - x3)
h = abs(y2 - y1)
```

По познатата ни от училище формула за намиране на **лице на триъгълник**, ще пресметнем лицето:

```
s = a * h / 2
```

Единственото, което остава, е да отпечатаме лицето на конзолата:

```
print(s)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#0>.

## Задача: пренасяне на тухли

Строителни работници трябва да пренесат общо **x тухли**. **Работниците** са **w** на брой и работят едновременно. Те превозват тухлите в колички, всяка с **вместимост m** тухли. Напишете програма, която прочита целите числа **x, w** и **m** и пресмята **колко най-малко курса** трябва да направят работниците, за да превозят тухлите.

## Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- **Броят тухли  $x$**  се чете от първия ред.
- **Броят работници  $w$**  се чете от втория ред.
- **Вместимостта на количката  $m$**  се чете от третия ред.

Всички входни числа са цели и в диапазона [1 ... 1000].

## Изход

Да се отпечата на конзолата **минималният брой курсове**, необходими за превозване на тухлите.

### Примерен вход и изход

Вход	Изход	Обяснения
120 2 30	2	Имаме <b>2</b> работника, всеки вози по <b>30</b> тухли на курс. Общо работниците возят по <b>60</b> тухли на курс. За да превозят <b>120</b> тухли, са необходими точно <b>2</b> курса.

Вход	Изход	Обяснения
355 3 10	12	Имаме <b>3</b> работника, всеки вози по <b>10</b> тухли на курс. Общо работниците возят по <b>30</b> тухли на курс. За да превозят <b>355</b> тухли, са необходими точно <b>12</b> курса: <b>11</b> пълни курса превозват <b>330</b> тухли и последният <b>12-ти</b> курс пренася последните <b>25</b> тухли.

Вход	Изход	Обяснения
5 12 30	1	Имаме <b>5</b> работника, всеки вози по <b>30</b> тухли на курс. Общо работниците возят по <b>150</b> тухли на курс. За да превозят <b>5</b> тухли, е достатъчен само <b>1</b> курс (макар и непълен, само с 5 тухли).

## Насоки и подсказки

Входът е стандартен, като единствено трябва да внимаваме за последователността, в която прочитаме данните:

```
x = int(input())
w = int(input())
m = int(input())
```

Пресмятаме колко **тухли** носят работниците на един курс:

```
bricks_in_course = w * m
```

Като разделим общия брой на **тухлите, пренесени за 1 курс**, ще получим броя **курсове**, необходими за пренасянето им. Ще използваме функцията **math.ceil(...)**, за да закръглим получения резултат нагоре. **Не забравяме** да добавим и **import math** в началото на работния файл, за да може функцията **math.ceil(...)** да работи. Когато тухлите могат да се пренесат с **точен брой курсове**, делението ще връща точно число и няма да има нищо за закръгляне. Съответно, когато не е така, резултатът от делението ще е **броя на точните курсове**, но с десетична част. Десетичната част ще се закръгли нагоре и така ще се получи нужният **1 курс** за оставащите тухли;

```
total_courses = math.ceil(x / bricks_in_course)
```

Накрая принтираме резултата на конзолата:

```
print(total_courses)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#1>.

## Задачи с единична проверка

Втората задача от практическия изпит по “Основи на програмирането” обхваща **условна конструкция и прости пресмятания**. Ето няколко примера:

### Задача: точка върху отсечка

Върху хоризонтална прива е разположена **хоризонтална отсечка**, зададена с **x** координатите на двата си края: **first** и **second**. Точка е разположена **върху** същата хоризонтална прива и е зададена с **x** координатата си. Напишете програма, която проверява дали точката е **вътре или вън** от отсечката и изчислява **разстоянието до по-близкия край** на отсечката.

#### Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- На първия ред стои числото **first** – **единия край на отсечката**.
- На втория ред стои числото **second** – **другия край на отсечката**.
- На третия ред стои числото **point** – **местоположението на точката**.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

#### Изход

Резултатът да се отпечата на конзолата:

- На първия ред да се отпечата "in" или "out" – дали точката е върху отсечката или извън нея.
- На втория ред да се отпечата разстоянието от точката до най-близкия край на отсечката.

### Примерен вход и изход

Вход	Изход	Визуализация
10 5 7	in 2	

Вход	Изход	Визуализация
8 10 5	out 3	

Вход	Изход	Визуализация
1 -2 3	out 2	

### Насоки и подсказки

Първо прочитаме входните данни:

```
first = int(input())
second = int(input())
point = int(input())
```

Тъй като не знаем коя **точка** е от ляво и коя е от дясно, ще си направим две променливи, които да ни отбелнязват това. Тъй като **левата точка** е винаги тази с по-малката **x координата**, ще ползваме функцията **min(...)**, за да я намерим. Съответно, **дясната** е винаги тази с по-голяма **x координата** – за нея ще ползваме **max(...)**. Ще намерим и разстоянието от **точката x** до **двете точки**. Понеже не знаем положението им една спрямо друга, ще използваме функцията **abs(...)**, за да получим положителен резултат:

```
left = min(first, second)
right = max(first, second)

distance_left = abs(left - point)
distance_right = abs(right - point)
```

По-малкото от двете **разстояния** ще намерим ползвайки **min(...)**:

```
min_distance = min(distance_left, distance_right)
```

Остава да намерим дали **точката** е на линията или извън нея. Точката ще се намира на **линията** винаги, когато тя **съвпада** с някоя от другите две точки или х координатата ѝ се намира **между тях**. В противен случай, точката се намира **извън линията**. След проверката изкарваме едното от двете съобщения ("in" или "out"), спрямо това кое условие е удовлетворено:

```
if left <= point <= right:
    print('in')
else:
    print('out')
```

Накрая принтираме **разстоянието**, намерено преди това:

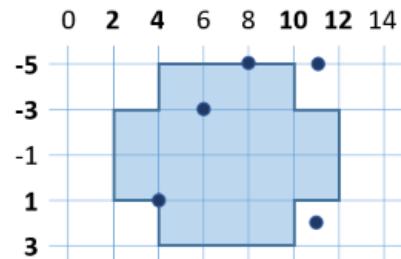
```
print(min_distance)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#2>.

### Задача: точка във фигура

Да се напише програма, която проверява дали дадена точка (с координати **x** и **y**) е **вътре** или **извън** следната фигура от картинката.



#### Вход

От конзолата се четат **две цели числа** (по едно на ред): **x** и **y**.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

#### Изход

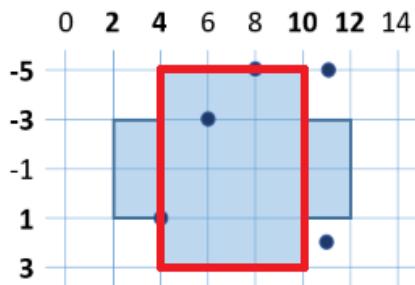
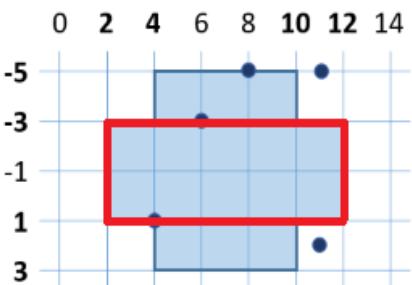
Да се отпечата на конзолата "in" или "out" – дали точката е **вътре** или **извън** фигурата (на контура е вътре).

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
8 -5	in	6 -3	in	11 -5	out	11 2	out

#### Насоки и подсказки

За да разберем дали **точката** е във **фигурата**, ще разделим **фигурата** на два **четириъгълника**:



Достатъчно условие е **точката** да се намира в един от тях (който и да е от двата или и в двета едновременно), за да се намира във **фигурата**.

Четем от конзолата входните данни:

```
x = int(input())
y = int(input())
```

Ще създадем две променливи, които ще отбелязват дали точката се намира в някой от правоъгълниците:

```
pointInRect1 = 2 <= x <= 12 and -3 <= y <= 1
pointInRect2 = 4 <= x <= 10 and -5 <= y <= 3
```

При отпечатването на съобщението ще проверим дали някоя от променливите е приела стойност **True**. Достатъчно е **само една** от тях да е **True**, за да се намира точката във фигурата:

```
if pointInRect1 or pointInRect2:
    print('in')
else:
    print('out')
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#3>.

## Задачи с по-сложни проверки

Третата задача на практическия изпит по “Основи на програмирането” включва няколко вложени проверки съчетани с прости пресмятания. Ето няколко примера:

### Задача: дата след 5 дни

Дадени са две числа **d** (ден) и **m** (месец), които формират **дата**. Да се напише програма, която отпечатва датата, която ще бъде **след 5 дни**. Например 5 дни след 28.03 е датата 2.04. Приемаме, че месеците: април, юни, септември и ноември имат по 30 дни, февруари има 28 дни, а останалите имат по 31 дни. Месеците да се отпечатват с **водеща нула**, когато са едноцифрови (например 01, 08).

## Вход

Входът се чете от конзолата и се състои от два реда:

- На първия ред стои едно цяло число **d** в интервала [1 ... 31] – ден. Номерът на деня не надвишава броя дни в съответния месец (напр. 28 за февруари).
- На втория ред стои едно цяло число **m** в интервала [1 ... 12] – месец. Месец 1 е януари, месец 2 е февруари, ..., месец 12 е декември. Месецът може да съдържа водеща нула (напр. април може да бъде изписан като 4 или 04).

## Изход

Отпечатайте на конзолата един единствен ред, съдържащ дата след 5 дни във формат **ден.месец**. Месецът трябва да бъде двуцифreno число с водеща нула, ако е необходимо. Денят трябва да е без водеща нула.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
28	2.04	27	1.01	25	30.01	26	3.03
03		12		1		02	

### Насоки и подсказки

Прочитаме входните данни от конзолата:

```
d = int(input())
m = int(input())
```

За да си направим по-лесно проверките, ще си създадем една променлива, която ще съдържа **броя дни**, които има в месеца, който сме задали:

```
days_in_month = 31
```

```
if m == 2:
    days_in_month = 28

if m == 4 or m == 6 or m == 9 or m == 11:
    days_in_month = 30
```

Увеличаваме **дения** с 5:

```
d += 5
```

Проверяваме дали **деният** не е станал по-голям от броя дни, които има в съответния **месец**. Ако това е така, трябва да извадим дните от месеца от получения ден, за да получим нашият ден на кой ден от следващия месец съответства:

```
if d > days_in_month:
    d -= days_in_month
```

След като сме минали в **следващия месец**, това трябва да се отбележи, като увеличим първоначално зададения месец с 1. Трябва да проверим, дали той не е станал по-голям от 12 и ако е така, да коригираме. Тъй като няма как да прескочим повече от **един месец**, когато увеличаваме с 5 дни, следната проверка е достатъчна:

```
if d > days_in_month:
    d -= days_in_month

m += 1

if m > 12:
    m = 1
```

Остава само да принтираме резултата на конзолата. Важно е да **форматираме изхода** правилно, за да се появява водещата нула в първите 9 месеца. Това става, като добавим **форматиращ стринг** **%02d** за втория елемент:

```
print('%d.%02d' % (d, m))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#4>.

## Задача: суми от 3 числа

Дадени са 3 цели числа. Да се напише програма, която проверява дали **сумата на две от числата е равна на третото**. Например, ако числата са 3, 5 и 2, сумата на две от числата е равна на третото:  $2 + 3 = 5$ .

### Вход

От конзолата се четат **три цели числа**, по едно на ред. Числата са в диапазона [1 ... 1000].

### Изход

- Да се отпечата на конзолата един ред, съдържащ решението на задачата във формат " $a + b = c$ ", където **a**, **b** и **c** са измежду входните три числа и  $a \leq b$ .
- Ако задачата няма решение, да се отпечата "No" на конзолата.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
3 5 2	$2 + 3 = 5$	2 2 4	$2 + 2 = 4$	1 1 5	No	2 6 3	No

### Насоки и подсказки

Приемаме входа от конзолата:

```
a = int(input())
b = int(input())
c = int(input())
```

Трябва да проверим дали **сумата** на някоя двойка числа е равна на третото. Имаме три възможни случая:

- $a + b = c$
- $a + c = b$
- $b + c = a$

Ще напишем **рамка**, която после ще допълним с нужния код. Ако никое от горните три условия не е изпълнено, ще зададем на програмата да принтира "No":

```
if a + b == c:
    # TODO

elif a + c == b:
    # TODO

elif b + c == a:
    # TODO

else:
    print('No')
```

Сега остава да разберем реда, в който ще се изписват **двете събираеми** на изхода на програмата. За целта ще направим **вложено условие**, което проверява кое от двете числа е по-голямото. При първия случай, ще стане по следния начин:

```
if a + b == c:
    if a > b:
        print('%d + %d = %d', b, a, c)

    else:
        print('%d + %d = %d', a, b, c)
```

Аналогично, ще допълним и другите два случая. Пълният код на проверките и изходът на програмата ще изглеждат така:

```
if a + b == c:
    if a > b:
        print('%d + %d = %d', b, a, c)
```

```

else:
    print('%d + %d = %d', a, b, c)

elif a + c == b:
    if a < c:
        print('%d + %d = %d', a, c, b)
    else:
        print('%d + %d = %d', c, a, b)

elif b + c == a:
    if b < c:
        print('%d + %d = %d', b, c, a)
    else:
        print('%d + %d = %d', c, b, a)

else:
    print('No')

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#5>.

## Задачи с единичен цикъл

Четвъртата задача от практическия изпит по “Основи на програмирането” включва единичен цикъл с пристап логика в него. Ето няколко примера:

### Задача: суми през 3

Дадени са  $n$  цели числа  $a_1, a_2, \dots, a_n$ . Да се пресметнат сумите:

- $sum1 = a_1 + a_4 + a_7 + \dots$  (сумират се числата, започвайки от първото със стъпка 3).
- $sum2 = a_2 + a_5 + a_8 + \dots$  (сумират се числата, започвайки от второто със стъпка 3).
- $sum3 = a_3 + a_6 + a_9 + \dots$  (сумират се числата, започвайки от третото със стъпка 3).

### Вход

Входните данни се четат от конзолата. На първия ред стои цяло число  $n$  ( $0 \leq n \leq 1000$ ). На следващите  $n$  реда са  $n$  цели числа в интервала  $[-1000 \dots 1000]$ :  $a_1, a_2, \dots, a_n$ .

### Изход

На конзолата трябва да се отпечатат 3 реда, съдържащи търсените 3 суми, във формат като в примерите.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2	sum1 = 3 sum2 = 5 sum3 = 0	4	sum1 = 19 sum2 = -2 sum3 = 6	5	sum1 = 10 sum2 = 13 sum3 = 2
3		7		3	
5		-2		5	
		6		2	
		12		7	
				8	

### Насоки и подсказки

Ще вземем **броя на числата** от конзолата и ще декларираме **начални стойности** на трите суми:

```
n = int(input())
sum1 = sum2 = sum3 = 0
```

Тъй като не знаем предварително колко числа ще обработваме, ще ги прочитаме едно по едно в **цикъл**, който ще се повтори **n** на **брой пъти** и ще ги обработваме в тялото на цикъла:

```
for i in range(0, n):
    a = int(input())

    # TODO
```

За да разберем в коя от **трите суми** трябва да добавим числото, ще разделим **поредния му номер** на **три** и ще използваме **остатъка**. Ще използваме променливата **i**, която следи **броя завъртания** на цикъла, за да разберем на кое пред число сме. Когато резултатът от **i % 3** е **нула**, това означава, че ще добавяме това число към **първата** сума, когато е **1** към **втората** и съответно, когато е **2** към **третата**:

```
for i in range(0, n):
    a = int(input())

    if i % 3 == 0:
        sum1 += a

    if i % 3 == 1:
        sum2 += a
```

```

if i % 3 == 2:
    sum3 += a

```

Накрая, ще отпечатаме резултата на конзолата в изисквания от условието формат:

```

print('sum1 = %d' % sum1)
print('sum2 = %d' % sum2)
print('sum3 = %d' % sum3)

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#6>.

## Задача: поредица от нарастващи елементи

Дадена е редица от  $n$  числа:  $a_1, a_2, \dots, a_n$ . Да се пресметне **дължината на най-дългата нарастваща поредица** от последователни елементи в редицата от числа.

### Вход

Входните данни се четат от конзолата. На първия ред стои цяло число  $n$  ( $0 \leq n \leq 1000$ ). На следващите  $n$  реда стоят  $n$  цели числа в интервала  $[-1000 \dots 1000]$ :  $a_1, a_2, \dots, a_n$ .

### Изход

На конзолата трябва да се отпечата едно число – **дължината** на най-дългата нарастваща редица.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
3		4		4		4	
5	2	2		1		5	
2		8	2	2	3	6	
4		7		4		7	
		6		4		8	

### Насоки и подсказки

За решението на тази задача трябва да помислим малко **по-алгоритично**. Дадена ни е **редица от числа** и трябва да проверяваме дали всяко **следващо** число е **поголямо от предходното** и ако е така да броим колко дълга е тази поредица, в която това условие е изпълнено. След това трябва да намерим **коя подредица** е **найдълга**. За целта, нека да си направим няколко променливи, които ще ползваме през хода на задачата:

```
n = int(input())
count_current_longest = count_longest = a_prev = 0
```

Променливата **n** е броя числа, които ще получим от конзолата. В **count\_current\_longest** ще запазваме броя на елементите в нарастваща подредица, която броим в момента. Напр. при редицата: 5, 6, 1, 2, 3 **count\_current\_longest** ще бъде 2, когато сме стигнали втория елемент от броенето (5, 6, 1, 2, 3) и ще стане 3, когато стигнем последния елемент (5, 6, 1, 2, 3), понеже нарастващата редица 1, 2, 3 има 3 елемента. Ще използваме **count\_longest**, за да запазим най-дългата нарастваща редица. Останалите променливи, който ще използваме, са **a** - числото, на което се намираме в момента, и **a\_prev** - предишното число, което ще сравним с **a**, за да разберем дали редицата расте.

Започваме да обхождаме числата и проверяваме дали настоящото число **a** е по-голямо от предходното **a\_prev**. Ако това е изпълнено, значи подредицата е нарастваща и трябва да увеличим броя ѝ с 1. Това запазваме в променливата, която следи дължината на редицата, в която се намираме в момента, а именно - **count\_current\_longest**. Ако числото **a** не е по-голямо от предходното, това означава, че започва нова подредица и трябва да стартираме броенето от 1. Накрая, след всички проверки, **a\_prev** става числото, което използваме в момента, и започваме цикъла от начало със следващото въведено **a**.

Ето и примерна реализация на описания алгоритъм:

```
for i in range(0, n):
    a = int(input())

    if a > a_prev:
        count_current_longest += 1
    else:
        count_current_longest = 1

    a_prev = a
```

Остава да разберем коя от всички редици е най-дълга. Това ще направим с проверка в цикъла дали редицата, в която се намираме в момента, е станала по-дълга от дължината на най-дългата намерена до сега. Целият цикъл ще изглежда така:

```
for i in range(0, n):
    a = int(input())

    if a > a_prev:
        count_current_longest += 1
    else:
        if count_current_longest > count_longest:
            count_longest = count_current_longest

    a_prev = a
```

```

    count_current_longest = 1

if count_current_longest > count_longest:
    count_longest = count_current_longest

a_prev = a

```

Накрая принтираме дължината на **най-дългата** намерена редица:

```
print(count_longest)
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#7>.

## Задачи за чертане на фигури на конзолата

Петата задача от практическия изпит по “Основи на програмирането” изисква използване на един или няколко вложени цикъла за рисуване на някаква фигурука на конзолата. Може да се изискват логически размишления, извършване на прости пресмятания и проверки. Задачата проверява способността на студентите да мислят логически и да измислят прости алгоритми за решаване на задачи, т.е. да мислят алгоритмично. Ето няколко примера за изпитни задачи:

### Задача: перфектен диамант

Да се напише програма, която прочита от конзолата цяло число **n** и чертае перфектен диамант с размер **n** като в примерите по-долу.

#### Вход

Входът е цяло число **n** в интервала [1 ... 1000].

#### Изход

На конзолата трябва да се отпечата диамантът като в примерите.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	<pre> * *- *</pre>	3	<pre> * *- *-*- *- *</pre>	5	<pre> * *- *-*- *-*-*- *-*-*-*- *-*-*-*- *-*-*- *- *</pre>	4	<pre> * *- *-*- *-*-*- *-*-*- *- *</pre>

## Насоки и подсказки

В задачите за чертане на фигурки най-важното, което трябва да преценим е **последователността**, в която ще рисуваме. Кои елементи се **повтарят** и с какви **стъпки**. В конкретната задача, ясно може да забележим, че **горната и долната** част на диаманта са **еднакви**. Най-лесно ще я решим, като направим **един цикъл**, който чертае **горната част**, и след това още **един**, който чертае **долната** (обратно на горната).

Първо прочитаме числото **n** от конзолата:

```
n = int(input())
```

Започваме да рисуваме **горната половина** на диаманта. Ясно виждаме, че **всеки ред** започва с няколко празни места и **\***. Ако се вгледаме по- внимателно, ще забележим, че **празните места** са винаги равни на **n - броя на реда** (на първия ред са  $n-1$ , на втория -  $n-2$  и т.н.) Ще започнем с това да нарисуваме броя **празни места**, както и **първата звездичка**. Нека не забравяме да използваме **print(..., end='')** вместо само **print(...)**, за да оставаме на **същия ред**. На края на реда пишем **print()**, за да преминем на **нов ред**. Забележете, че започваме да броим от **1**, а не от **0**. След това ще остане само да добавим няколко пъти **-\***, за да **довършим** реда.

Ето фрагмент от кода за начертаване на **горната част** на диаманта:

```
for i in range(1, n + 1):
    print(' ' * (n - i)) + '*', end='')

# TODO: Draw the rest of the line

print()
```

Остава да **довършим** **всеки ред** с нужния брой **-\*** елементи. На всеки ред трябва да добавим **i - 1** такива **елемента** (на първия  $1-1 \rightarrow 0$ , на втория  $\rightarrow 1$  и т.н.)

Ето и пълният код за начертаване на **горната част** на диаманта:

```
for i in range(1, n + 1):
    print(' ' * (n - i)) + '*', end='')

    for j in range(0, i - 1):
        print('-*', end='')

    print()
```

За да изрисуваме **долната част** на диаманта, трябва да обърнем **горната** на обратно. Ще броим от **n - 1**, тъй като ако започнем от **n**, ще изрисуваме средния ред два пъти. Не забравяйте да добавите **reversed(...)** на **range(...)** на главния цикъл.

Ето го и кода за начертаване на **долната част** на диаманта:

```

for i in reversed(range(1, n)):
    print(" " * (n - i) + "*", end='')

for j in range(1, i):
    print('* ', end='')

print()

```

Остава да сглобим цялата програма като първо четем входа, печатаме горната част на диаманта и след него и долната част на диаманта.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#8>.

## Задача: правоъгълник със звездички в центъра

Да се напише програма, която прочита от конзолата цяло число **n** и чертае правоъгълник с размер **n** с две звездички в центъра, като в примерите по-долу.

### Вход

Входът е цяло число **n** в интервала [2 ... 1000].

### Изход

На конзолата трябва да се отпечата правоъгълникът като в примерите.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	%%% %**% %%%	3	%%%%%% % % % ** % % % %%%%%%	4	%%%%%%%/ % % % ** % % % %%%%%%	5	%%%%%%%%% % % % % % ** % % % %%%%%%%%

### Насоки и подсказки

Прочитаме входните данни от задачата:

```
n = int(input())
```

Първото нещо, което лесно забелязваме, е че **първият и последният ред** съдържат **2 \* n** символа **%**. Ще започнем с това и после ще нарисуваме средата на четириъгълника:

```

print('*' * (2 * n))

# TODO: Draw middle of the rectangle

print('*' * (2 * n))

```

От дадените примери виждаме, че **средата** на фигурата винаги има **нечетен брой** редове. Забелязваме, че когато е зададено **четно число**, броят на редовете е равен на **предишното нечетно** ( $2 \rightarrow 1, 4 \rightarrow 3$  и т.н.). Създаваме променлива, която представлява броя редове, които ще има нашият правоъгълник, и я коригираме, ако числото **n** е **четно**. След това ще нарисуваме **правоъгълника без звездичките**. Всеки ред има за **начало и край** символа **%** и между тях  **$2 * n - 2$**  празни места (ширината е  **$2 * n$**  и вадим 2 за двета процента в края). Не забравяйте да преместите кода за **последния ред след цикъла**:

```

num_rows = n

if n % 2 == 0:
    num_rows -= 1

for i in range(0, num_rows):
    print('*', end='')

# TODO: Place stars and spaces

    print('*', end='')

```

Можем да **стартираме и тестваме** кода до тук. Всичко без двете звездички в средата трябва да работи коректно.

Сега остава **в тялото** на цикъла да добавим и **звездичките**. Ще направим проверка дали сме на **средния ред**. Ако сме на средния (това ще проверим като сравним **i** с **floor(num\_rows / 2)**), ще рисуваме **реда** заедно **със звездичките**, ако не - ще рисуваме **нормален ред**. Не забравяйте да добавите **from math import floor** в началото на работния файл, за да може да използваме функцията **floor(...)**. Редът със звездичките има **n - 2** **празни места** (**n** е половината дължина и махаме звездичката и процента), **две звезди** и отново **n - 2** **празни места**. Двата процента в началото и в края на реда ги оставяме извън проверката:

```

for i in range(0, num_rows):
    print('*', end='')

    if i == floor(num_rows / 2):
        print(' ' * (n - 2) + '**' + (' ' * (n - 2)))

```

```

else:
    print(' ' * (n * 2 - 2))

print('%', end=' ')

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1059#9>.

## Задачи с вложени цикли с по-сложна логика

Последната (шеста) задача от практическия изпит по “Основи на програмирането” изисква използване на **няколко вложени цикъла и по-сложна логика в тях**. Задачата проверява способността на студентите да мислят алгоритично и да решават нетривиални задачи, изискващи съставянето на цикли. Следват няколко примера за изпитни задачи.

### Задача: четворки нарастващи числа

По дадена двойка числа **a** и **b** да се генерират всички четворки **n1, n2, n3, n4**, за които **a ≤ n1 < n2 < n3 < n4 ≤ b**.

#### Вход

Входът съдържа две цели числа **a** и **b** в интервала [0 ... 1000], по едно на ред.

#### Изход

Изходът съдържа всички търсени **четворки числа**, подредени в нарастващ ред, по една на ред.

#### Примерен вход и изход

Вход	Изход
3	3 4 5 6
7	3 4 5 7 3 4 6 7 3 5 6 7 4 5 6 7

Вход	Изход
5	
7	No

Вход	Изход
10 13	10 11 12 13

#### Насоки и подсказки

Ще прочетем входните данни от конзолата. Създаваме и допълнителната променлива **count**, която ще следи дали има **съществуваща редица** числа:

```

a = int(input())
b = int(input())

```

```
count = 0
```

Най-лесно ще решим задачата, ако логически я разделим на части. Ако се изисква да изведем всички редици от едно число между **a** и **b**, ще го направим с **един цикъл**, който изкарва всички числа от **a** до **b**. Нека помислим как ще стане това с **редици от две числа**. Отговорът е лесен - ще ползваме **вложени цикли**:

```
for i in range(a, b + 1):
    for j in range(i + 1, b + 1):
        print('%d %d' % (i, j))
```

Можем да тестваме недописаната програма, за да проверим дали работи коректно до този момент. Тя трябва да отпечатва всички двойки числа **i, j**, за които **i ≤ j**.

Тъй като всяко **следващо число** от редицата трябва да е **по-голямо от предишното**, вторият цикъл ще се върти от **i + 1** (следващото по-голямо число). Съответно, ако **не съществува редица** от две нарастващи числа (**a** и **b** са равни), вторият цикъл **няма да се изпълни** и няма да се разпечата нищо на конзолата.

**Аналогично**, остава да реализираме по същия начин **вложените цикли** и за **четири** числа. Ще добавим и **увеличаване на брояча** (**count**), който инициализирахме в началото, за да знаем дали **съществува** такава **редица**:

```
for i in range(a, b + 1):
    for j in range(i + 1, b + 1):
        for k in range(j + 1, b + 1):
            for l in range(k + 1, b + 1):
                print('%d %d %d %d' % (i, j, k, l))
                count += 1
```

Накрая ще проверим дали **броячът** е равен на 0 и съответно ще принтираме "No" на конзолата, ако е така:

```
if count == 0:
    print('No')
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1059#10>.

## Задача: генериране на правоъгълници

По дадено число **n** и **минимална площ** **m** да се генерират всички правоъгълници с цели координати в интервала **[-n ... n]**, с площ поне **m**. Генерираните правоъгълници да се отпечатат в следния формат:

**(left, top) (right, bottom) -> area**

Правоъгълниците се задават чрез горния си ляв и долния си десен ъгъл. В сила са следните неравенства:

- $-n \leq \text{left} < \text{right} \leq n$
- $-n \leq \text{top} < \text{bottom} \leq n$

## Вход

От конзолата се въвеждат две числа, по едно на ред:

- Цяло число **n** в интервала [1 ... 100] – задава минималната и максималната координата на връх.
- Цяло число **m** в интервала [0 ... 50 000] – задава минималната площ на генерираните правоъгълници.

## Изход

- На конзолата трябва да се отпечатат описаните правоъгълници във формат като в примерите по-долу.
- Ако за числата **n** и **m** няма нито един правоъгълник, да се изведе "No".
- Редът на извеждане на правоъгълниците е без значение.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2 17	No	3 36	(-3, -3) (3, 3) -> 36	1 2	(-1, -1) (0, 1) -> 2 (-1, -1) (1, 0) -> 2 (-1, -1) (1, 1) -> 4 (-1, 0) (1, 1) -> 2 (0, -1) (1, 1) -> 2

## Насоки и подсказки

Да прочетем входните данни от конзолата. Отново ще създадем и един **брояч**, в който ще пазим броя на намерените правоъгълници:

```
n = int(input())
m = int(input())

count = 0
```

Изключително важно е да успеем да си представим задачата, преди да започнем да я решаваме. В нашия случай се изисква да търсим правоъгълници в координатна система. Нещото, което знаем е, че **лявата точка** винаги ще има координата **x**, **по-малка от дясната**. Съответно **горната** винаги ще има **по-малка** координата **y** от **долната**. За да намерим всички правоъгълници, ще трябва да направим **цикъл**, подобен на този от предходната задача, но този път **не всеки следващ цикъл** ще започва от **следващото число**, защото някои от **координатите** може да са **равни** (например **left** и **top**):

```

for left in range(-n, n):
    for top in range(-n, n):
        for right in range(left + 1, n + 1):
            for bottom in range(top + 1, n + 1):
                # TODO

```

С променливите **left** и **right** ще следим координатите по **хоризонталата**, а с **top** и **bottom** - по **вертикалата**. Важното тук е да знаем кои координати кои са, за да можем да изчислим правилно страните на правоъгълника. Сега трябва да намерим **лицето на правоъгълника** и да направим проверка дали то е **по-голямо** или **равно** на **m**. Едната страна ще е **разликата между left и right**, а другата - **между top и bottom**. Тъй като координатите евентуално може да са разменени, ще ползваме **абсолютни стойности**. Отново добавяме и **брояча** в цикъла, като броим **само четириъгълниците**, които изписваме. Важно е да забележим, че поредността на изписване е **left, top, right, bottom**, тъй като така е зададено в условието:

```

for left in range(-n, n):
    for top in range(-n, n):
        for right in range(left + 1, n + 1):
            for bottom in range(top + 1, n + 1):
                area = abs(right - left) * abs(bottom - top)

                if area >= m:
                    print('(%d, %d) (%d, %d) -> %d'
                          % (left, top, right, bottom, area))

                count += 1

```

Накрая принтираме "No", ако не съществуват такива правоъгълници:

```

if count == 0:
    print('No')

```

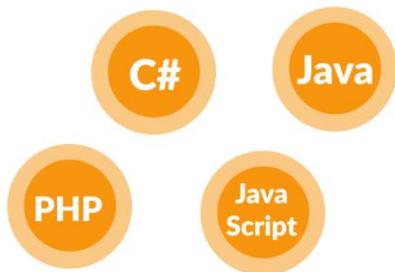
## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1059#11>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтуни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтуни работи пряко с компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 8.2. Подготовка за практически изпит – част II

В настоящата глава ще разгледаме един **практически изпит по основи на програмирането**, проведен в СофтУни на 18 декември 2016 г. Задачите дават добра представа какво можем да очакваме на приемния изпит по програмиране в СофтУни. Изпитът покрива изучавания учебен материал от настоящата книга и от курса "Programming Basics" в СофтУни.

## Видео

Гледайте видео-урок по тази глава: [https://youtube.com/watch?v=LprL\\_zAnz-o](https://youtube.com/watch?v=LprL_zAnz-o).

## Изпитни задачи

Традиционно приемният изпит в СофтУни се състои от **6 практически задачи по програмиране**:

- Задача с прости сметки (без проверки).
- Задача с единична проверка.
- Задача с по-сложни проверки.
- Задача с единичен цикъл.
- Задача с вложени цикли (чертане на фигурка на конзолата).
- Задача с вложени цикли и по-сложна логика.

Да разгледаме една **реална изпитна тема**, задачите в нея и решенията им.

## Задача: разстояние

Напишете програма, която да пресмята **колко километра изминава кола**, за която знаем **първоначалната скорост** (км/ч), **времето** в минути, след което **увеличава скоростта с 10%**, **второ време**, след което **намалява скоростта с 5%**, и **времето до края на пътуването**. За да намерите **разстоянието** трябва да **превърнете минутите в часове** (например 70 минути = 1.1666 часа).

## Входни данни

От конзолата се четат **4 реда**:

- Първоначалната скорост в км/ч – цяло число в интервала [1 ... 300].
- Първото време в минути – цяло число в интервала [1 ... 1000].
- Второто време в минути – цяло число в интервала [1 ... 1000].
- Третото време в минути – цяло число в интервала [1 ... 1000].

## Изходни данни

Да се отпечата на конзолата едно число: **изминатите километри**, форматирани до втория символ след десетичния знак.

## Примерен вход и изход

Вход	Изход	Обяснения
90 60 70 80	330.90	<p>Разстояние с първоначална скорост: <math>90 \text{ км/ч} * 1 \text{ час (60 мин)} = 90 \text{ км}</math></p> <p>След увеличението: <math>90 + 10\% = 99.00 \text{ км/ч} * 1.166 \text{ часа (70 мин)} = 115.50 \text{ км}</math></p> <p>След намаляването: <math>99 - 5\% = 94.05 \text{ км/ч} * 1.33 \text{ часа (80 мин)} = 125.40 \text{ км}</math></p> <p>Общо изминати: 330.9 км</p>

Вход	Изход	Обяснения
140 112 75 190	917.12	<p>Разстояние с първоначална скорост: <math>140 \text{ км/ч} * 1.86 \text{ часа (112 мин)} = 261.33 \text{ км}</math></p> <p>След увеличението: <math>140 + 10\% = 154.00 \text{ км/ч} * 1.25 \text{ часа (75 мин)} = 192.5 \text{ км}</math></p> <p>След намаляването: <math>154.00 - 5\% = 146.29 \text{ км/ч} * 3.16 \text{ часа (190 мин)} = 463.28 \text{ км}</math></p> <p>Общо изминати: 917.1166 км</p>

## Насоки и подсказки

Вероятно е подобно условие да изглежда на пръв поглед **объркващо** и непълно, което **придава** допълнителна **сложност** на една лесна задача. Нека **разделим** заданието на няколко **подзадачи** и да се опитаме да **решим** всяка една от тях, което ще ни отведе и до крайния резултат:

- Прочитане на входните данни.
- Изпълнение на основната програмна логика.
- Пресмятане и оформяне на крайния резултат.

Съществената част от програмната логика се изразява в това да пресметнем какво ще бъде **изминатото разстояние след всички промени** в скоростта. Тъй като по време на **изпълнението** на програмата, част от данните, с които разполагаме, се променят, то бихме могли да **разделим решението** на няколко логически обособени стъпки:

- Пресмятане на изминатото **разстояние** с първоначална скорост.
- Промяна на **скоростта** и пресмятане на изминатото **разстояние**.
- Последна промяна на **скоростта** и пресмятане.

- Сумиране.

По условие **входните данни** се въвеждат на **четири** отделни реда, по тази причина следва да извикаме функцията **input()** четири пъти:

```
initial_speed_string = input()
# first_interval_string = TODO
# second_interval_string = TODO
# third_interval_string = TODO
```

Следва да **преобразуваме входните данни** в подходящи **типове**, за да можем да извършим необходимите пресмятания. Избираме да използваме тип **int**, тъй като в условието на задачата е упоменато, че входните данни ще бъдат в определен **интервал**, за който този тип данни е напълно достатъчен. Преобразуването извършваме по следния начин:

```
initial_speed = int(initial_speed_string)
# first_interval = TODO
# second_interval = TODO
# third_interval = TODO
```

По този начин успяхме да се справим успешно с **първата подзадача** - приемане на входните данни.

Първоначално **запазваме една променлива**, която ще използваме многократно. Този подход на централизация ни дава **гъвкавост и възможност да променяме** цялостния резултат на програмата с минимални усилия. В случай, че се наложи да променим стойността, трябва да го направим само на **едно място в кода**, което ни спестява време и усилия:

```
minutes_per_hour = 60
```



**Избягването на повтарящ се код** (централизация на програмната логика) в задачите, които разглеждаме в настоящата книга, изглежда на пръв поглед излишна, но този подход е от съществено значение при изграждането на мащабни приложения в реална работна среда и упражняването му в начален стадий на изучаване само ще подпомогне усвояването на един качествен стил на програмиране.

**Изминалото време** в часове пресмятаме като **разделим** подаденото ни **време на 60** (минутите в един час). **Изминатото разстояние** намираме като **умножим** **началната скорост с изминалото време** (в часове). След това променяме скоростта, като я увеличаваме с **10%** (по условие). Пресмятането на **процентите**, както и следващите изминати **разстояния**, извършваме по следния начин:

- **Интервалът от време** (в часове) намираме като **разделим** зададения интервал в минути на минутите, които се съдържат в един час (60).

- Изминатото разстояние намираме като умножим интервала (в часове) по скоростта, която получаваме след увеличението.
- Следващата стъпка е да намалим скоростта с 5%, както е зададено по условие.
- Намираме оставащото разстояние по описания начин в първите две точки.

```
first_interval_hours = first_interval / minutes_per_hour
first_distance = initial_speed * first_interval_hours

speed_after_increase = initial_speed + ((initial_speed * 10) / 100)
second_interval_hours = second_interval / minutes_per_hour
second_distance = speed_after_increase * second_interval_hours
```

До този момент успяхме да изпълним две от най-важните подзадачи, а именно приемането на данните и тяхната обработка. Остава ни само да пресметнем крайния резултат. Тъй като по условие се изисква той да бъде форматиран до 2 символа след десетичния знак, можем да направим това по следния начин:

```
final_distance = \
    first_distance + second_distance + third_distance
final_result = ("{:.2f}".format(final_distance))
print(final_result)
```

В случай, че сте работили правилно и изпълните програмата с входните данни от условието на задачата, ще се уверите, че тя работи коректно.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#0>.

## Задача: смяна на плочки

Хараламби има събрани пари, с които иска да смени плочките на пода в банята. Като подът е правоъгълник, а плочките са триъгълни. Напишете програма, която да пресмята дали събраните пари ще му стигнат. Подават се широчината и дължината на пода, както и едната страна на триъгълника с височината към нея. Трябва да пресметнете колко плочки са нужни, за да се покрие пода. Броят на плочките трябва да се закръгли към по-високо цяло число и да се прибавят още 5 броя за фирма. В допълнение ни се подават още – цената на плочка и сумата за работата на майстор.

### Входни данни

Като параметри на функцията подаваме 7 числа:

- Събраните пари.
- Широчината на пода.

- Дължината на пода.
- Страната на триъгълника.
- Височината на триъгълника.
- Цена на една плочка.
- Сумата за майстора.

Всички числа са реални числа в интервала [0.00 ... 5000.00].

## Изходни данни

На конзолата трябва да се отпечата на **един ред**:

- Ако парите **са достатъчно**:
  - "{Оставащите пари} lv left."
- Ако парите **НЕ са достатъчно**:
  - "You'll need {Недостигащите пари} lv more."

Резултатът трябва да е **форматиран до втория символ** след десетичния знак.

## Примерен вход и изход

Вход	Изход	Обяснения
1000		Площ на пода $\rightarrow 5.55 * 8.95 = 49.67249$
5.55		Площа на плочка $\rightarrow 0.9 * 0.85 / 2 = 0.3825$
8.95	You'll need	Необходими плочки $\rightarrow 49.67249 / 0.3825 = 129.86\dots$
0.90	1209.65 lv	$= 130 + 5$ фирма $= 135$
0.85	more.	Обща сума $\rightarrow 135 * 13.99 + 321$ (майстор) $= 2209.65$
13.99		$2209.65 > 1000 \rightarrow$ не достигат 1209.65 лева
321		

Вход	Изход	Обяснения
500		Площ на пода $\rightarrow 3 * 2.5 = 7.5$
3		Площа на плочка $\rightarrow 0.5 * 0.7 / 2 = 0.175$
2.5		Необходими плочки $\rightarrow 7.5 / 0.175 = 42.857\dots = 43 +$
0.5	25.60 lv	5 фирма $= 48$
0.7	left.	Обща сума $\rightarrow 48 * 7.8 + 100$ (майстор) $= 474.4$
7.80		$474.4 < 500 \rightarrow$ остават 25.60 лева
100		

## Насоки и подсказки

Следващата задача изисква от нашата програма да приема повече входни данни и извърши по-голям брой изчисления, въпреки че решението е **идентично**.

Приемането на данните от потребителя извършваме по добре **познатия ни начин**. Обърнете внимание, че в раздел **Вход** в условието е упоменато, че всички входни данни ще бъдат **реални числа** и поради тази причина бихме използвали **float**.

След като вече разполагаме с всичко необходимо, за да изпълним програмната логика, можем да пристъпим към следващата част. Как бихме могли да **изчислим** какъв е **необходимият** брой плочки, които ще бъдат достатъчни за покриването на целия под? Условието, че плочките имат **триъгълна** форма, би могло да доведе до объркване, но на практика задачата се свежда до съвсем **прости изчисления**. Бихме могли да пресметнем каква е **общата площ на пода** по формулата за намиране на площ на правоъгълник, както и каква е **площта на една плочка** по съответната формула за триъгълник.

За да пресметнем какъв **брой плочки** са необходими, **разделяме площта на пода на площта на една плочка**(като не забравяме да прибавим 5 допълнителни броя плочки, както е по условие).



Обърнете внимание, че в условието е упоменато да закръглим броя на плочките, получен от делението, до по-високо цяло число, след което да прибавим 5. Потърсете повече информация за системната функционалност за това: **math.ceil(...)**.

До крайния резултат можем да стигнем, като **пресметнем общата сума**, която е необходима, за да бъде покрит целия под, като **съберем цената на плочките с цената за майстора**, която имаме от входните данни. Можем да се досетим, че **общият разход** за плочките можем да получим, като **умножим броя плочки по цената за една плочка**. Дали сумата, с която разполагаме, ще бъде достатъчна, разбираме като сравним събраните до момента пари (от входните данни) и общите разходи:

```
if budget >= total_cost:
    # TODO: Print message
else:
    # TODO: Print message
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#1>.

## Задача: магазин за цветя

Магазин за цветя предлага 3 вида цветя: **хризантеми**, **рози** и **лалета**. Цените зависят от сезона.

Сезон	Хризантеми	Рози	Лалета
пролет / лято	2.00 лв./бр.	4.10 лв./бр.	2.50 лв./бр.
есен / зима	3.75 лв./бр.	4.50 лв./бр.	4.15 лв./бр.

В празнични дни цените на всички цветя се **увеличават с 15%**. Предлагат се следните **отстъпки**:

- За закупени повече от 7 лалета през пролетта – **5% от цената** на целия букет.
- За закупени 10 или повече рози през зимата – **10% от цената** на целия букет.
- За закупени повече от 20 цветя общо през всички сезони – **20% от цената** на целия букет.

Отстъпките се правят по така написания ред и могат да се наслагват! Всички отстъпки важат след осъществяването за празничен ден!

Цената за аранжиране на букета винаги е **2 лв.** Напишете програма, която изчислява **цената за един букет**.

## Входни данни

Входът се чете от **конзолата** и съдържа **точно 5 реда**:

- **Броят** на закупените **хризантеми** – цяло число в интервала [0 ... 200].
- **Броят** на закупените **рози** – цяло число в интервала [0 ... 200].
- **Броят** на закупените **лалета** – цяло число в интервала [0 ... 200].
- **Сезонът** – [Spring, Summer, Autumn, Winter].
- **Дали денят е празник** – [Y - да / N - не].

## Изходни данни

Да се отпечата на конзолата 1 число – **цената на цветята**, форматирана до втория символ след десетичния знак.

## Примерен вход и изход

Вход	Изход	Обяснения
2 4 8 Spring Y	46.14	Цена: $2*2.00 + 4*4.10 + 8*2.50 = 40.40$ лв. Празничен ден: $40.40 + 15\% = 46.46$ лв. <b>5% намаление</b> за повече от 7 лалета през пролетта: 44.14 Общо цветята са 20 или по-малко: <b>няма намаление</b> $44.14 + 2$ <b>за аранжиране</b> = 46.14 лв.

Вход	Изход	Обяснения
3 10 9	69.39	Цена: $3*3.75 + 10*4.50 + 9*4.15 = 93.60$ лв. Не е празничен ден: няма увеличение <b>10% намаление</b> за 10 или повече рози през зимата: 84.24

Вход	Изход	Обяснения
Winter N		Общо цветята са повече от 20: с <b>20% намаление</b> = 67.392 $67.392 + 2$ <b>за аранжиране</b> = 69.392 лв.

## Насоки и подсказки

След като прочитаме внимателно условието разбираме, че отново се налага да извършваме **прости пресмятания**, но с разликата, че този път ще са необходими и **повече логически проверки**. Следва да обърнем повече **внимание** на това, в какъв момент се **извършват промените** по крайната цена, за да можем правилно да изградим логиката на нашата програма. Отново, удебеленият текст ни дава достатъчно **насоки** как да подходим. Като за начало, отделяме вече **декларирани** стойности в **променливи**, както направихме и в предишните задачи:

```
# Initial price list
rose_autumn_winter_price = 4.50
rose_spring_summer_price = 4.10
tulip_autumn_winter_price = 4.15
tulip_spring_summer_price = 2.50
chrysanthemum_autumn_winter_price = 3.75
chrysanthemum_spring_summer_price = 2.00
arrange_price = 2.00
```

Правим същото и за останалите вече декларирани стойности:

```
# Price increases
price_increase_percentage = 15

# Price decreases
tulip_price_decrease_percentage = 5
rose_price_decrease_percentage = 10
total_price_decrease_percentage = 20

# Price decrease thresholds
tulip_price_decrease_threshold = 7
rose_price_decrease_threshold = 10
total_price_decrease_threshold = 20
```

Следващата ни подзадача е да **прочетем** правилно **входните** данни от конзолата. Подхождаме по добре познатия ни начин, но този път **комбинираме** две отделни функции – една за **прочитане** на ред от конзолата и друга за **преобразуването** му в числен тип данни:

```
chrysanthemums_purchased = int(input())
# roses_purchased = TODO
# tulips_purchased = TODO
# season = TODO
# is_special_day = TODO
```

Нека помислим кой е най-подходящият начин да **структуриме** нашата програмна логика. От условието става ясно, че пътят на програмата се разделя основно на две части: **пролет / лято и есен / зима**. Разделението ще направим с условна конструкция **if-else**, като преди това заделяме променливи за **цените** на отделните цветя, както и за **крайния резултат**:

```
if season == 'Winter' or season == 'Autumn':
    roses_price = roses_purchased * rose_autumn_winter_price
    chrysanthemums_price = chrysanthemums_purchased * \
                           chrysanthemum_autumn_winter_price
    tulips_price = tulips_purchased * tulip_autumn_winter_price
    total_costs = roses_price
                  + chrysanthemums_price + tulips_price
else:
    # TODO
```

Остава ни да извършим **няколко проверки** относно **намаленията** на различните видове цветя, в зависимост от сезона, и да модифицираме крайния резултат.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#2>.

## Задача: оценки

Напишете програма, която да **пресмята статистика на оценки** от изпит. В началото програмата получава **броя на студентите**, явили се на изпита, и за **всеки студент неговата оценка**. На края програмата трябва да **изпечата процента на студенти** с оценка между 2.00 и 2.99, между 3.00 и 3.99, между 4.00 и 4.99, 5.00 или повече, както и **средни успех** на изпита.

### Входни данни

От конзолата се четат **поредица** от числа, всяко на отделен ред:

- На първия ред – **броя на студентите**, явили се на изпит – цяло число в интервала [1 ... 1000].
- За **всеки един студент** на отделен ред – **оценката от изпита** – реално число в интервала [2.00 ... 6.00].

### Изходни данни

Да се отпечатат на конзолата **5 реда**, които съдържат следната информация:

- "Top students: {процент студенти с успех 5.00 или повече}%".
- "Between 4.00 and 4.99: {между 4.00 и 4.99 включително}%".
- "Between 3.00 and 3.99: {между 3.00 и 3.99 включително}%".
- "Fail: {по-малко от 3.00}%".
- "Average: {среден успех}".

Резултатите трябва да са **форматирани до втория символ** след десетичния знак.

## Примерен вход и изход

Вход	Изход	Обяснения
10		
3.00		
2.99		
5.68	Top students: 30.00%	5 и повече – <b>трима</b> = 30% от 10
3.01	Between 4.00 and 4.99: 30.00%	Между 4.00 и 4.99 – <b>трима</b> = 30% от 10
4	Between 3.00 and 3.99: 20.00%	Между 3.00 и 3.99 – <b>двама</b> = 20% от 10
4	Fail: 20.00%	Под 3 – <b>двама</b> = 20% от 10
6.00	Average: 4.06	Средният успех е: $3 + 2.99 + 5.68 + 3.01 + 4 + 4 + 6 + 4.50 + 2.44 + 5 = 40.62 / 10 = 4.062$
4.50		
2.44		
5		

Вход	Изход
6	
2	Top students: 33.33%
3	Between 4.00 and 4.99: 16.67%
4	Between 3.00 and 3.99: 16.67%
5	Fail: 33.33%
6	Average: 3.70
2.2	

## Насоки и подсказки

От условието виждаме, че **първо** ще ни бъде подаден **броя** на студентите, а едва **след това оценките** им. По тази причина **първо** ще приемем **броя** на студентите, като конвертираме прочетената стойност до **int**. За да прочетем и обработим самите оценки, ще използваме **for** цикъл. Всяка итерация на цикъла ще прочита и обработва по една оценка:

```
number_of_students = int(input())
for i in range(number_of_students):
    # TODO
```

Преди да се изпълни кода от **for** цикъла заделяме променливи, в които ще пазим броя на студентите за всяка група: слаби резултати (до 2.99), резултати от 3 до 3.99, от 4 до 4.99 и оценки над 5. Ще ни е необходима и още една променлива, в която да пазим сумата на всички оценки, с помощта на която ще изчислим средната оценка на всички студенти.

```
number_of_failed_students = 0
number_of_average_students = 0
number_of_good_students = 0
number_of_excellent_students = 0
total_result = 0
```

Завъртаме цикъла и в него декларираме още една променлива, в която ще запазваме текущата въведена оценка. Променливата ще е от тип **float** и на всяка итерация проверяваме каква е стойността ѝ. Според тази стойност увеличаваме броя на студентите в съответната група с 1, като не забравяме да увеличим и общата сума на оценките, която също следим.

```
for i in range(number_of_students):
    grade = float(input())
    total_result += grade
    if grade < 3:
        number_of_failed_students += 1
    else:
        # TODO: check other groups
```

Какъв процент заема дадена група студенти от общия брой, можем да пресметнем като умножим броя на студентите от съответната група по 100 и след това разделим на общия брой студенти.

**Крайният** резултат оформяме по добре познатия ни начин до втория символ след десетичния знак.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#3>.

## Задача: коледна шапка

Да се напише програма, която прочита от конзолата цяло число **n** и чертае коледна шапка с ширина  $4 * n + 1$  колони и височина  $2 * n + 5$  реда като в примерите по-долу.

## Входни данни

Входът се чете от конзолата – едно **цяло число n** в интервала [3 ... 100].

## Изходни данни

Да се отпечата на конзолата **коледна шапка**, точно както в примерите.

### Примерен вход и изход

Вход	Изход	Вход	Изход
4	<pre>...../ \..... .....\ /..... ....***..... ....*-*-*.... ....*--*--*... ....*---*---*.. ..*----*----*.. .----*----*.. *-----*-----* ***** *.*.*.*.*.*.* *****</pre>	7	<pre>...../ \..... .....\ /..... ....***..... ....*-*-*.... ....*--*--*... ....*---*---*.. ..*----*----*.. .----*----*.. *-----*-----* ...*-----*-----*.. .----*----*.. *-----*-----* ***** *.*.*.*.*.*.*. *****</pre>

## Насоки и подсказки

При задачите за **чертане** на конзолата, най-често потребителят въвежда **едно цяло число**, което е свързано с **общата големина на фигурката**, която трябва да начертаем. Тъй като в условието е упоменато как се изчисляват общата дължина и широчина на фигурката, можем да ги използваме за **отправни точки**. От примерите ясно се вижда, че без значение какви са входните данни, винаги имаме **първи два реда**, които са с почти идентично съдържание.

```
:::::::<|>:::::::
```

Забелязваме също така, че **последните три реда** винаги присъстват, **два** от които са напълно **еднакви**.

```
*****
*.*.*.*.*.*.*.
*****
```

От тези наши наблюдения можем да изведем **формулата за височина на променливата част** на коледната шапка. Използваме зададената по условие формула за общата височина, като изваждаме големината на непроменливата част. Получаваме  **$(2 * n + 5) - 5$**  или  **$2 * n$** .

За **начертаването на динамичната** част от фигурката ще използваме **цикъл**. Размерът на цикъла ще бъде от **Одо широчината**, която имаме по условие, а именно  **$4 * n + 1$** . Тъй като тази формула ще използваме на **няколко места** в кода, е добра практика да я изнесем в **отделна променлива**. Преди изпълнението на цикъла би следвало да **заделим променливи за броя** на отделните символи, които участват в динамичната част: **точки и тирета**. Чрез изучаване на примерите можем да изведем формули и за **стартовите стойности** на тези променливи. Първоначално **ти retata** са **0**, но броя на **точките** ясно се вижда, че можем да получим като от **общата широчина** извадим **3** (броя символи, които изграждат върха на коледната шапка) и след това **разделим на 2**, тъй като броя точки от двете страни на шапката е еднакъв.

```
.....***.....
.....*-*-*.....
.....*--*--*.....
....*---*---*....
...*---*---*...
..*---*---*..
.*---*---*..
*---*---*
```

Остава да изпълним тялото на цикъла, като **след всеки** начертан ред **намаляваме** броя на точките с **1**, а **ти retata увеличим** с **1**. Нека не забравяме да начертаем и по една **звездичка** между тях. Последователността на чертане в тялото на цикъла е следната:

- Символен низ от точки
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от точки

В случай че сме работили правилно получаваме фигурки, идентични на тези от примерите.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#4>.

## Задача: комбинации от букви

Напишете програма, която да принтира на конзолата **всички комбинации от 3 букви** в зададен интервал, като се пропускат комбинациите, **съдържащи зададена от конзолата буква**. Накрая трябва да се принтира броят отпечатани комбинации.

### Входни данни

Входът се чете от конзолата и съдържа **точно 3 реда**:

- Малка буква от английската азбука за начало на интервала – от 'a' до 'z'.
- Малка буква от английската азбука за край на интервала – от **първата буква** до 'z'.
- Малка буква от английската азбука – от 'a' до 'z' – като комбинациите, съдържащи тази буква се пропускат.

### Изходни данни

Да се отпечатат на един ред **всички комбинации**, отговарящи на условието, следвани от **броя им**, разделени с интервал.

### Примерен вход и изход

Вход	Изход	Обяснения
a c b	aaa aac aca acc caa cac cca ccc 8	Всички възможни комбинации с буквите 'a', 'b' и 'c' са: aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc Комбинациите, <b>съдържащи 'b'</b> , не са валидни. Остават <b>8</b> валидни комбинации.

Вход	Изход
f k h	fff ffg ffi ffj ffk fgf fgg fgi fgj fgk fif fig fii fij fik fjj fjk fkf fkg fki fkj fkk gff gfg gfi gfj gfk ggf ggg ggi ggj ggk gif gig gii gjj gik gif gig gjj gjk gkf gkg gki gkj gkk iff ifg ifi ifj ifk ifg iff iif ifg iig iii iij iik ijf ijj iji ijj ijk ikf ikg iki ijk ikk jff jfg jfi jfj jfk jgf jgg jgi jgj jgk jif jig jii jjj jik jjf jjg jji jjj jjk jkf jkg jki jkj jkk kff kfg kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kfj kjg kji kjy kjk kkf kkg kki kkj kkk 125

Вход	Изход
a c z	aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc 27

## Насоки и подсказки

За последната задача имаме по условие входни данни на **3 реда**, които са представени от по един символ от **ASCII таблицата** (<http://www.asciitable.com/>). Бихме могли да използваме вече дефинираната функция в езика Python, **ord(...)**, чрез която ще получим ASCII кода на подадения символ:

```
start_letter = ord(input()[0])
# TODO
```

Нека помислим как бихме могли да стигнем до **крайния резултат**. В случай че условието на задачата е да се принтират всички от началния до крайния символ (с пропускане на определена буква), как бихме постъпили?

Най-лесният и удачен начин е да използваме **цикъл**, с който да преминем през **всички символи** и открием тези, които са **различни от буквата**, която трябва да пропуснем. В езика Python можем да обходим всички символи от 'a' до 'z' по следния начин:

```
for i in range(ord('a'), ord('z') + 1):
    print(chr(i), end=' ')
```

Функцията **chr(...)** ще конвертира подадения ASCII код в символ. Резултатът от изпълнението на горния код е всички букви от **a** до **z** включително, принтирани на един ред и разделени с интервал. Това прилича ли на крайния резултат от нашата задача? Трябва да измислим **начин**, по който да се принтират по **3 символа**, както е по условие, вместо по **1**. Изпълнението на програмата много прилича на игрална машина. Там най-често печелим, ако успеем да наредим няколко еднакви символа. Да речем, че на машината имаме места за три символа. Когато **спрем** на даден **символ** на първото място, на останалите две места **продължават** да се изреждат символи от всички възможни. В нашия случай **всички възможни** са буквите от началната до крайната такава, зададена от потребителя, а решението на нашата програма е идентично на начина, по който работи игралната машина.

Използваме **цикъл**, който минава през **всички символи** от началната до крайната буква включително. На **всяка итерация** на **първия цикъл** пускаме **втори** със същите параметри (но **само ако** буквата на първия цикъл е валидна, т.е. не съвпада с тази, която трябва да изключим по условие). На всяка итерация на **втория цикъл** пускаме още **един** със **същите параметри** и същата **проверка**. По този начин ще имаме три вложени цикъла, като в тялото на **последния** ще добавяме символите към крайния резултат:

```
for i in range(ord(start_letter), ord(end_letter) + 1):
    if chr(i) != except_letter:
        # TODO
```

Нека не забравяме, че се изиска от нас да принтираме и **общия брой валидни комбинации**, които сме намерили, както и че те трябва да се принтират на **същия ред**, разделени с интервал. Тази подзадача оставяме на читателя.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1060#5>.

# Глава 9.1. Задачи за шампиони – част I

В настоящата глава ще предложим на читателя няколко малко по-трудни задачи, които имат за цел развиване на алгоритмични умения и усвояване на програмни техники за решаване на задачи с по-висока сложност.

## По-сложни задачи върху изучавания материал

Ще решим заедно няколко задачи по програмиране, които обхващат изучавания в книгата учебен материал, но по трудност надвишават обичайните задачи от приемните изпити в СофтУни. Ако искате да станете шампиони по основи на програмирането, ви препоръчваме да тренирате решаване на подобни по-сложни задачи, за да ви е лесно на изпитите.

### Задача: пресичащи се редици

Имаме две редици:

- Редица на Трибоначи (по аналогия с редицата на Фиbonачи), където всяко число е сумата от предните три (при дадени начални три числа).
- Редица, породена от **числова спирала**, дефинирана чрез обхождане като спирала (дясно, долу, ляво, горе, дясно, долу, ляво, горе и т.н.) на матрица от числа, стартирайки от нейния център с дадено начално число и стъпка на увеличение, със записване на текущите числа в редицата всеки път, когато направим завой.

Да се напише програма, която намира първото число, което се появява **и в двете** така дефинирани редици.

### Пример

Нека **редицата на Трибоначи** да започне с 1, 2 и 3. Това означава, **първата редица** че ще съдържа числата 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, 778, 1431, 2632, 4841, 8904, 16377, 30122, 55403, 101902 и т.н.

Същевременно, нека **числата в спиралата** да започнат с 5 и спиралата да се увеличава с 2 на всяка стъпка. Тогава **втората редица** ще съдържа числата 5, 7, 9, 13, 17, 23, 29, 37 и т.н. Виждаме, че 37 е първото число, което се среща в редицата на Трибоначи и в спиралата и това е търсеното решение на задачата.

Тогава **втората редица** ще съдържа числата 5, 7, 9, 13, 17, 23, 29, 37 и т.н. Виждаме, че 37 е първото число, което се среща в редицата на Трибоначи и в спиралата и това е търсеното решение на задачата.

45	...	...	...	...	...	55
...	17	19	21	23	...	...
...	15	5	7	...	...	...
...	13	11	9	...	...	...
37	...	...	...	...	29	...
...	...	...	...	...	...	65

## Входни данни

Входните данни трябва да бъдат прочетени от конзолата.

- На първите три реда от входа ще прочетете **три цели числа**, представляващи **първите три числа** в редицата на Трибоначи.
- На следващите два реда от входа, ще прочетете **две цели числа**, представляващи **първото число и стъпката** за всяка клетка на матрицата за спиралата от числа.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да ги проверявате.

## Изходни данни

Резултатът трябва да бъде принтиран на конзолата.

На единствения ред от изхода трябва да принтирате **най-малкото число**, което се среща и в двете последователности. Ако няма число в **диапазона [1 ... 1 000 000]**, което да се среща и в двете последователности, принтирайте "No".

## Ограничения

- Всички числа във входа ще бъдат в диапазона **[1 ... 1 000 000]**.
- Позволено работно време за програмата: 1.5 секунди.
- Позволена памет: 16 MB.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1		13		99		1	
2		25		99		4	
3	37	99	13	99	No	7	23
5		5		2		23	
2		2		2		3	

## Насоки и подсказки

Задачата изглежда доста сложна и затова ще я разбием на по-прости подзадачи: обработване на входа, генериране на редица на Трибоначи, генериране на числовата спирала и намиране на най-малкото общо число за двете редици.

## Обработване на входа

Първата стъпка от решаването на задачата е да прочетем и обработим входа. Входните данни се състоят от **5 цели числа**: **3** за редицата на Трибоначи и **2** за числовата спирала:

```

tribonacci_first = int(input())
tribonacci_second = int(input())
tribonacci_third = int(input())

spiral_current = int(input())
spiral_step = int(input())

```

След като имаме входните данни, трябва да помислим как ще генерираме числата в двете редици.

## Генериране на редица на Трибоначи

За редицата на Трибоначи всеки път ще събираме предишните три стойности и след това ще отнемваме стойностите на тези числа (трите предходни) с една позиция напред в редицата, т.е. стойността на първото трябва да приеме стойността на второто и т.н. Когато сме готови с числото, ще запазваме стойността му в **масив**. Понеже в условието на задачата е казано, че числата в редиците не превишават 1,000,000, можем да спрем генерирането на тази редица именно при 1,000,000:

```

tribonacci_numbers = [tribonacci_first,
                      tribonacci_second,
                      tribonacci_third]

tribonacci_current = tribonacci_third
while tribonacci_current < 1000000:
    tribonacci_current = tribonacci_first + \
                          tribonacci_second + \
                          tribonacci_third
    tribonacci_numbers.append(tribonacci_current)

    tribonacci_first = tribonacci_second
    tribonacci_second = tribonacci_third
    tribonacci_third = tribonacci_current

```

## Генериране на числова спирала

Трябва да измислим **зависимост** между числата в числовата спирала, за да можем лесно да генерираме всяко следващо число, без да се налага да разглеждаме матрици и тяхното обхождане. Ако разгледаме внимателно картиката от условието, можем да забележим, че **на всеки 2 "завоя"** в спиралата числата, които прескачаме, се увеличават **с 1**, т.е. от 5 до 7 и от 7 до 9 не се прескача нито 1 число, а директно **събираме със стъпката** на редицата. От 9 до 13 и от 13 до 17 прескачаме едно число, т.е. събираме два пъти стъпката. От 17 до 23 и от 23 до 29 прескачаме две числа, т.е. събираме три пъти стъпката и т.н.

Така виждаме, че при първите две имаме **последното числото + 1 \* стъпката**, при следващите две събираме с **2 \* стъпката** и т.н. Всеки път, когато искаме да стигнем до следващото число от спиралата, ще трябва да извършваме такива изчисления:

```
spiral_current += spiral_step * spiral_step_mul
```

Това, за което трябва да се погрижим, е **на всеки две числа нашият множител** (нека го наречем "кофициент") **да се увеличава с 1** (`spiral_step_mul++`), което може да се постигне с прости проверки за четност (`spiral_count % 2 == 0`). Целият код от генерирането на спиралата в **масив** е даден по-долу:

```
spiral_numbers = [spiral_current]
spiral_count = 0
spiral_step_mul = 1

while spiral_current < 1000000:
    spiral_current += spiral_step * spiral_step_mul
    spiral_numbers.append(spiral_current)
    spiral_count += 1

    if spiral_count % 2 == 0:
        spiral_step_mul += 1
```

### Намиране на общо число за двете редици

След като сме генерирали числата и в двете редици, можем да пристъпим към обединението им и изграждането на крайното решение. Как ще изглежда то? За **всяко от числата** в едната редица (започвайки от по-малкото) ще проверяваме дали то съществува в другата. Първото число, което отговаря на този критерий ще бъде **отговорът** на задачата.

Търсенето във втория масив ще направим **линейно**, а за по-любопитните ще оставим да си го оптимизират, използвайки техниката наречена **двоично търсене** (Binary Search), тъй като вторият масив се генерира сортиран, т.е. отговаря на изискването за прилагането на този тип търсене. Кодът за намиране на нашето решение ще изглежда така:

```
found = False
for i in range(0, len(tribonacci_numbers)):
    for j in range(0, len(spiral_numbers)):
        if tribonacci_numbers[i] == spiral_numbers[j] \
            and tribonacci_numbers[i] <= 1000000:
            print(tribonacci_numbers[i])
            found = True
            break
    if found:
        break
```

```
if not found:
    print("No")
```

Решението на задачата използва масиви за запазване на стойностите. Масивите не са необходими за решаването на задачата. Съществува алтернативно решение, което генерира числата и работи директно с тях, вместо да ги записва в масив. Можем на всяка стъпка да проверяваме дали числата от двете редици съвпадат. Ако това е така, ще принтираме на конзолата числото и ще прекратим изпълнението на нашата програма. В противен случай, ще видим текущото число на коя редица е по-малко и ще генерираме следващото, там където "изоставаме". Идеята е, че ще генерираме числа от редицата, която е "по-назад", докато не прескочим текущото число на другата редица и след това обратното, а ако междувременно намерим съвпадение, ще прекратим изпълнението:

```
while tribonacci_current <= 1000000
    and spiral_current <= 1000000:
    if tribonacci_current == spiral_current:
        print("TODO: Print and stop execution")
    elif tribonacci_current < spiral_current:
        print("TODO: Generate next Tribonacci number")
    else:
        print("TODO: Generate next Spiral number")
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1061#0>.

## Задача: магически дати

Дадена е **дата** във формат "дд-мм-гггг", напр. 26-09-2018. Изчисляваме **теглото на тази дата**, като вземем всичките ѝ цифри, умножим всяка цифра с останалите след нея и накрая съберем всички получени резултати. В нашия случай имаме 8 цифри: 17032007, така че теглото е  $1*7 + 1*0 + 1*3 + 1*2 + 1*0 + 1*0 + 1*7 + 7*0 + 7*3 + 7*2 + 7*0 + 7*0 + 7*7 + 0*3 + 0*2 + 0*0 + 0*0 + 0*7 + 3*2 + 3*0 + 3*0 + 3*7 + 2*0 + 2*0 + 2*7 + 0*0 + 0*7 + 0*7 = 144$ .

Нашата задача е да напишем програма, която намира всички **магически дати** - дати между две определени години (включително), отговарящи на дадено във входните данни тегло. Датите трябва да бъдат принтирани в нарастващ ред (по дата) във формат "дд-мм-гггг". Ще използваме само валидните дати в традиционния календар (високосните години имат 29 дни през февруари).

## Входни данни

Входните данни трябва да бъдат прочетени от конзолата. Състоят се от 3 реда:

- Първият ред съдържа цяло число: **начална година**.
- Вторият ред съдържа цяло число: **краяна година**.
- Третият ред съдържа цяло число: **търсеното тегло** за датите.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да се проверяват.

## Изходни данни

Резултатът трябва да бъде принтиран на конзолата, като последователни дати във **формат "дд-мм-гггг"**, подредени по дата в нарастващ ред. Всяка дата трябва да е на отделен ред. В случай, че няма съществуващи магически дати, да се принтира "No".

## Ограничения

- Началната и крайната година са цели числа в периода [1900 - 2100].
- Магическото тегло е цяло число в диапазона [1 ... 1000].
- Позволено работно време за програмата: 0.75 секунди.
- Позволена памет: 16 MB.

## Примерен вход и изход

Вход	Изход
2007	17-03-2007
2007	13-07-2007
144	31-07-2007

Вход	Изход
2003	
2004	No
1500	

Вход	Изход
2011	01-01-2011
2012	10-01-2011
14	01-10-2011
	10-10-2011

Вход	Изход
	09-01-2013
	17-01-2013
	23-03-2013
	11-07-2013
	01-09-2013
2012	10-09-2013
2014	09-10-2013
80	17-10-2013
	07-11-2013
	24-11-2013
	14-12-2013
	23-11-2014
	13-12-2014
	31-12-2014

## Насоки и подсказки

Започваме с вмъкване на необходимите функционалност и с четенето от входните данни от конзолата. В случая имаме вмъкваме типа **date** (с ново име **datetype**), функциите **timedelta** и **floor** и четем 3 цели числа:

```
from datetime import date as datatype
from datetime import timedelta
from math import floor

first_year = int(input())
second_year = int(input())
number_to_search_for = int(input())
```

Разполагайки с началната и крайната година, е хубаво да разберем как ще минем през всяка дата, без да се объркваме от това колко дена има в месеца и дали годината е високосна и т.н.

## Обхождане на всички дати

За обхождането ще се възползваме от функционалността, която ни дава **date** типът в Python. Ще си дефинираме **променлива за началната дата**, което можем да направим, използвайки "конструктора", който приема година, месец и ден. Знаем, че годината е началната година, която сме получили като параметър, а месеца и деня трябва да са съответно януари и 1-ви:

```
date = datatype(first_year, 1, 1)
```

След като имаме началната дата, искаме да направим **цикъл**, който се изпълнява, докато не превишим **крайната година** (или докато не преминем 31 декември в крайната година, ако сравняваме целите дати), като на всяка стъпка увеличава с по 1 ден.

За да увеличаваме с 1 ден при всяко завъртане, ще използваме метода **timedelta()** в Python за създаването на 1 ден разлика, която да добавим към **date**. Добавянето на 1 ден към датата ще помогне Python да се грижи вместо нас за прескачането в следващия месец и година, да следи колко дни има даден месец и да се погрижи вместо нас за високосните години:

```
date = date + timedelta(days=1)
```

В крайна сметка нашият цикъл ще изглежда по следния начин:

```
while date.year <= second_year:
```

```
# TODO: Check the date for number_to_search_for

date = date + timedelta(days=1)
```

## Пресмятане на теглото

Всяка дата се състои от точно 8 символа (цифри) - 2 за **дения** (**d1, d2**), 2 за **месеца** (**d3, d4**) и 4 за **годината** (**d5** до **d8**). Това означава, че всеки път ще имаме едно и също пресмятане и може да се възползваме от това, за **да дефинираме формулата статично** (т.е. да не обикаляме с цикли, реферирайки различни цифри от датата, а

да изпишем цялата формула). За да успеем да я изпишем, ще ни трябват **всички цифри от датата** в отделни променливи, за да направим всички нужни умножения. Използвайки операциите деление и взимане на остатък върху отделните компоненти на датата, чрез **day**, **month** и **year**, можем да извлечем всяка цифра. Трябва да внимаваме с целочисленото деление на 10 (**/ 10**), което няма да е целочислено тук и затова след всяко целочислено деление ще закръглеме изрично до най-малкото цяло число, чрез метода **floor(...)**:

```
d1 = floor(date.day / 10)      # First day digit
d2 = date.day % 10              # Second day digit

d3 = floor(date.month / 10)     # First month digit
d4 = date.month % 10            # Second month digit

d5 = floor(date.year / 1000)    # First year digit
d6 = floor(date.year / 100) % 10 # Second year digit
d7 = floor(date.year / 10) % 10 # Third year digit
d8 = date.year % 10             # Fourth year digit
```

Нека обясним и един от по-интересните редове тук. Нека вземе за пример взимането на втората цифра от годината (**d6**). При нея делим годината на 100 и взимаме остатък от 10. Какво постигаме така? Първо с деленето на 100 отстраняваме последните 2 цифри от годината (пример: **2018 / 100 = 20**). С остатъка от деление на 10 взимаме последната цифра на полученото число (**20 % 10 = 0**) и така получаваме 0, което е втората цифра на 2018.

Остава да направим изчислението, което ще ни даде магическото тегло на дадена дата. За да не изписваме всички умножения, както е показано в примера, ще приложим просто **групиране**. Това, което трябва да направим, е да умножим всяка цифра с тези, които са след нея. Вместо да изписваме **d1 \* d2 + d1 \* d3 + ... + d1 \* d8**, може да съкратим този израз до **d1 \* (d2 + d3 + ... + d8)**, следвайки математическите правила за групиране, когато имаме умножение и събиране. Прилагайки същото опростяване за останалите умножения, получаваме следната формула:

```
weight = d1 * (d2 + d3 + d4 + d5 + d6 + d7 + d8) + \
         d2 * (d3 + d4 + d5 + d6 + d7 + d8) + \
         d3 * (d4 + d5 + d6 + d7 + d8) + \
         d4 * (d5 + d6 + d7 + d8) + \
         d5 * (d6 + d7 + d8) + \
         d6 * (d7 + d8) + \
         d7 * d8
```

## Отпечатване на изхода

След като имаме пресметнато теглото на дадена дата, трябва да проверим дали съвпада с търсеното от нас магическо тегло, за да знаем, дали трябва да се

принтира или не. Проверката може да се направи със стандартен **if** оператор, като трябва да се внимава при принтирането датата да е в правилния формат. Защастие имаме вече всяка една от цифрите, които трябва да отпечатаме, а именно **d1** до **d8**. Тук трябва да се внимава с типовете данни. Тъй като конкатенацията на низове и операцията събиране на числа се извършват с един и същ оператор, трябва да конвертираме цифрите към низове:

```
if weight == number_to_search_for:
    print(str(d1) + str(d2) + "-" + str(d3) + str(d4) +
          "-" + str(d5) + str(d6) + str(d7) + str(d8))
found = True
```

**Внимание:** тъй като обхождаме датите от началната към крайната, те винаги ще бъдат подредени във възходящ ред, както е по условие.

И накрая, ако не сме намерили нито една дата, отговаряща на условията, ще имаме **False** стойност във **found** променливата и ще можем да отпечатаме "**No**":

```
if not found:
    print("No")
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1061#1>.

## Задача: пет специални букви

Дадени са две числа: **начало** и **край**. Напишете програма, която генерира всички комбинации от 5 букви, всяка измежду множеството `{'a', 'b', 'c', 'd', 'e'}`, така че "теглото" на тези 5 букви да е число в интервала **[начало ... край]**, включително. Принтирайте ги по азбучен ред, на един ред, разделени с интервал.

Теглото на една буква се изчислява по следния начин:

```
weight('a') = 5;
weight('b') = -12;
weight('c') = 47;
weight('d') = 7;
weight('e') = -32;
```

Теглото на редицата от букви **c1, c2, ..., cn** е изчислено, като се премахват всички букви, които се повтарят (от дясно наляво), и след това се пресметне формулата:

$$\text{weight}(c_1, c_2, \dots, c_n) = 1 * \text{weight}(c_1) + 2 * \text{weight}(c_2) + \dots + n * \text{weight}(c_n)$$

Например, теглото на **bcdcc** се изчислява по следния начин:

Първо премахваме повтарящите се букви и получаваме **bcd**. След това прилагаме формулата:  **$1 * \text{weight}('b') + 2 * \text{weight}('c') + 3 * \text{weight}('d') = 1 * (-12) + 2 * 47 + 3 * 7 = 103$** .

Друг пример:  **$\text{weight}("cadae") = \text{weight}("cade") = 1 * 47 + 2 * 5 + 3 * 7 + 4 * (-32) = -50$** .

## Входни данни

Входните данни се четат от конзолата. Състоят се от два реда:

- Числото за **начало** е на първия ред.
- Числото за **край** е на втория ред.

Входните данни винаги ще бъдат валидни и винаги ще са в описанния формат. Няма нужда да се проверяват.

## Изходни данни

Резултатът трябва да бъде принтиран на конзолата като поредица от низове, подредени по азбучен ред. Всеки низ трябва да бъде отделен от следващия с едно разстояние. Ако теглото на нито един от 5-буквените низове не съществува в зададения интервал, да се принтира "No".

## Ограничения

- Числата за **начало** и **край** ще бъдат цели числа в диапазона [-10000 ... 10000].
- Позволено работно време за програмата: 0.75 секунди.
- Позволена памет: 16 MB.

## Примерен вход и изход

Вход	Изход	Обяснения
40		
42	bcead bdcea	$\text{weight}("bcead") = 41$ $\text{weight}("bdcea") = 40$

Вход	Изход
300	
400	No

Вход	Изход
-1 1	bcdea cebda eaaad eaada eaadd eaade eaaed eadaa eadad eadae eadda eaddd eadde eadea eaded eadee eaead eaeda eaedd eaede eaeed eeaad eeda eead eeade eeaed eeead

Вход	Изход
200 300	baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc ebdac edbdc

## Насоки и подсказки

Като всяка задача, започваме решението с **обработване на входните данни**:

```
first_number = int(input())
second_number = int(input())
```

В задачата имаме няколко основни момента - **генерирането на всички комбинации** с дължина 5 включващи 5-те дадени букви, **премахването на повтарящите се букви** и **пресмятането на теглото** за дадена вече опростена дума. Отговорът ще се състои от всяка дума, чието тегло е в дадения интервал [**first\_number**, **second\_number**].

### Генериране на всички комбинации

За да генерираме **всички комбинации с дължина 1** използвайки 5 символа, бихме използвали **цикъл от 0...4**, като всяко число от цикъла ще искаме да отговаря на един символ. За да генерираме **всички комбинации с дължина 2** използвайки 5 символа (т.е. "aa", "ab", "ac", ..., "ba", ...), бихме направили **два вложени цикъла**, **всеки обхождащ цифрите от 0 до 4**, като отново ще направим, така че всяка цифра да отговаря на конкретен символ. Тази стъпка ще повторим 5 пъти, така че накрая да имаме **5 вложени цикъла** с индекси **i1, i2, i3, i4** и **i5**:

```
for i1 in range(0, 5):
    for i2 in range(0, 5):
        for i3 in range(0, 5):
            for i4 in range(0, 5):
                for i5 in range(0, 5):
```

Имайки всички 5-цифрени комбинации, трябва да намерим начин да "превърнем" петте цифри в дума с буквите от 'a' до 'e'. Един от начините да направим това е, като си **предефинираме прост стринг съдържащ буквите**, които имаме:

```
pattern = "abcde"
```

и за всяка цифра **взимаме буквата от конкретната позиция**. По този начин числото 00000 ще стане "aaaaa", числото 02423 ще стане "acecd". Можем да направим стринга от 5 букви по следния начин:

```
full_word = pattern[i1] + \
            pattern[i2] + \
            pattern[i3] + \
            pattern[i4] + \
            pattern[i5]
```

**Друг начин:** можем да преобразуваме цифрите до букви, използвайки подредбата им в ASCII таблицата. Изразът **chr(ord('a') + i)** ще ни даде резултата '**a**' при **i = 0**, '**b**' при **i = 1**, '**c**' при **i = 2** и т.н.

Така вече имаме генериирани всички 5-буквени комбинации и можем да продължим със следващата част от задачата.

**Внимание:** тъй като сме подбрали **pattern**, съобразен с азбучната подредба на буквите и циклите се въртят по подходящ начин, алгоритъмът ще генерира думите в азбучен ред и няма нужда от допълнително сортиране преди извеждане.

### Премахването на повтарящи се букви

След като имаме вече готовия низ, трябва да премахнем всички повтарящи се символи. Ще направим тази операция, като **добавяме буквите от ляво надясно в нов низ и всеки път преди да добавим буква ще проверяваме дали вече я има** - ако я има ще я пропускаме, а ако я няма ще я добавяме. За начало ще добавим първата буква към началния стринг:

```
word = pattern[i1]
```

След това ще направим същото и с останалите 4, проверявайки всеки път дали ги има със следното условие и метода **find(...)**. Това може да стане с цикъл по **full\_word** (оставяме това на читателя за упражнение), а може да стане и по мързеливия начин с copy-paste:

```
word = pattern[i1]
```

```
if word.find(pattern[i2]) == -1:
    word += pattern[i2]
if word.find(pattern[i3]) == -1:
    word += pattern[i3]
if word.find(pattern[i4]) == -1:
    word += pattern[i4]
if word.find(pattern[i5]) == -1:
    word += pattern[i5]
```

Методът **find(...)** връща индекса на конкретния елемент, ако бъде намерен или **-1**, ако елементът не бъде намерен. Следователно всеки път, когато получим **-1**, ще означава, че все още нямаме тази буква в новия низ с уникални букви и можем да я добавим, а ако получим стойност различна от **-1**, ще означава, че вече имаме буквата и няма да я добавяме.

### Пресмятане на теглото

Пресмятането на теглото е просто **обхождане на уникалната дума (word)**, получена в миналата стъпка, като за всяка буква трябва да вземем теглото ѝ и да я умножим по позицията. За всяка буква в обхождането трябва да пресметнем с каква стойност ще умножим позицията ѝ, например чрез използването на поредица от **if** конструкции:

```

multiplier = 0
if word[i] == 'a':
    multiplier = 5
if word[i] == 'b':
    multiplier = -12
if word[i] == 'c':
    multiplier = 47
if word[i] == 'd':
    multiplier = 7
if word[i] == 'e':
    multiplier = -32

```

След като имаме стойността на дадената буква, следва да я **умножим по позицията ѝ**. Тъй като индексите в стринга се различават с 1 от реалните позиции, т.е. индекс 0 е позиция 1, индекс 1 е позиция 2 и т.н., ще добавим 1 към индексите.

```
weight += multiplier * (i + 1)
```

Всички получени междинни резултати трябва да бъдат добавени към **общата сума за всяка една буква от 5-буквената комбинация**.

## Оформяне на изхода

Дали дадена дума трябва да се принтира, се определя по нейната тежест. Трябва ни условие, което да определи дали **текущата тежест е в интервала [начало ... край]**, подаден ни на входа в началото на програмата. Ако това е така, принтираме **пълната дума (full\_word)**.

**Внимавайте** да не принтирате думата от уникални букви. Тя ни бе необходима само за пресмятане на тежестта!

Думите са **разделени с интервал** и ще ги натрупваме в междинна променлива **result**, която е дефинирана като празен низ в началото:

```

if first_number <= weight <= second_number:
    result += full_word + " "

```

## Финални щрихи

Условието е изпълнено с **изключение** случаите, в които нямаме нито една дума в **подадения интервал**. За да разберем дали сме намерили такава дума, можем просто да проверим дали низът **result** има началната си стойност (а именно празен низ), ако е така - отпечатваме **"No"**, иначе печатаме целия низ без последния интервал (използвайки метода **.strip(...)**):

```
if result == "":
    print("No")
else:
    print(result.strip())
```

За по-любознателните читатели ще оставим за домашно да оптимизират работата на циклите като измислят начин да намалят броя итерации на вътрешните цикли.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1061#2>.

# Глава 9.2. Задачи за шампиони – част II

В тази глава ще разгледаме още три задачи, които причисляваме към категорията "за шампиони", т.е. по-трудни от стандартните задачи в тази книга.

## По-сложни задачи върху изучавания материал

Преди да преминем към конкретните задачи, трябва да поясним, че те могат да се решат по-лесно с **допълнителни знания за програмирането и езика Python** (функции, масиви, колекции, рекурсия и т.н.), но всяко едно решение, което ще дадем сега, ще използва единствено материал, покрит в тази книга. Целта е да се научим да съставяме **по-сложни алгоритми** на базата на сегашните ни знания.

### Задача: дни за страстно пазаруване

Лина има истинска страст за пазаруване. Когато тя има малко пари, веднага отива в първия голям търговски център (мол) и се опитва да изхарчи възможно най-много за дрехи, чанти и обувки. Но любимото ѝ нещо са зимните намаления. Нашата задача е да анализираме странното ѝ поведение и да **изчислим покупките**, които Лина прави, когато влезе в мола, както и **парите, които ѝ остават**, когато приключи с пазаруването си.

На **първия ред** от входа ще бъде подадена **сумата**, която Лина има **преди** да започне да пазарува. След това при получаване команда **"`mall.Enter`"**, Лина влиза в мола и започва да пазарува, докато не получи команда **"`mall.Exit`"**. Когато Лина започне да пазарува, **на всяка линия** от входа ще получите стрингове, които представляват **действия**, които **Лина изпълнява**. Всеки **символ** в стринга представлява **покупка или друго действие**. Стинговите команди могат да съдържат само символи от **ASCII таблицата**. ASCII кода на всеки знак има **връзка с това колко Лина трябва да плати** за всяка стока. Интерпретирайте символите по следния начин:

- Ако символът е **главна буква**, Лина получава **50% намаление**, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **малка буква**, Лина получава **70% намаление**, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **"%"**, Лина прави **покупка**, която намалява парите ѝ на половина.
- Ако символът е **"\*"**, Лина **изтегля пари от дебитната си карта** и добавя към наличните си средства 10 лева.
- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на символа от ASCII таблицата от наличните ѝ средства.

Ако някоя от стойностите на покупките е **по-голяма** от текущите налични средства, Лина **НЕ** прави покупката. Парите на Лина **не могат да бъдат по-малко от 0**.

Пазаруването завършва, когато се получи команда "**mall.Exit**". Когато това стане, трябва да се **принтират броя на извършени покупки и парите**, които са останали на Лина.

## Входни данни

Входните данни трябва да се четат от конзолата. На **първия ред** от входа ще бъде подадена **сумата**, която Лина има **преди да започне да пазарува**. На всеки следващ ред ще има определена команда. Когато получите команда "**mall.Enter**", на всеки следващ ред ще получавате стрингове, съдържащи **информация относно покупките / действията**, които Лина иска да направи. Тези стрингове ще продължат да бъдат подавани, докато не се получи команда "**mall.Exit**".

Командите "**mall.Enter**" и "**mall.Exit**" ще се подават само по веднъж.

## Изходни данни

Когато пазаруването приключи, на конзолата трябва да се принтира определен изход в зависимост от това какви покупки са били направени:

- Ако не са били направени **някакви покупки** – "No purchases. Money left: {останали пари} lv."
- Ако е направена **поне една покупка** – "{брой покупки} purchases. Money left: {останали пари} lv."

Парите трябва да се принтират с **точност от 2 символа** след десетичния знак.

## Ограничения

- Парите са число с **плаваща запетая** в интервала: [0 -  $7.9 \times 10^{28}$ ].
- Броят стрингове между "**mall.Enter**" и "**mall.Exit**" ще в интервала: [1 - 20].
- Броят символи във всеки стринг, който представлява команда, ще е в интервала: [1 - 20].
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

## Примерен вход и изход

Вход	Изход	Коментар
110 mall.Enter	1 purchases. Money left: 80.00 lv.	'd' има ASCII код 100. 'd' е малка буква и за това Лина получава 70% отстъпка. $100\% - 70\% = 30$ . $110 - 30 = 80$ лв.

Вход	Изход	Коментар
d mall.Exit		

Вход	Изход
110 mall.Enter % mall.Exit	1 purchases. Money left: 55.00 lv.

Вход	Изход
100 mall.Enter Ab ** mall.Exit	2 purchases. Money left: 58.10 lv.

## Насоки и подсказки

Ще разделим решението на задачата на три основни части:

- **Обработка** на входа.
- **Алгоритъм** на решаване.
- **Форматиране** на изхода.

Нека разгледаме всяка една част в детайли.

### Обработване на входа

Входът за нашата задача се състои от няколко компонента:

- На **първия ред** имаме **всички пари**, с които Лина ще разполага за пазаруването.
- На **всеки следващ ред** ще имаме някакъв вид **команда**.

Първата част от прочитането е тривиална:

```
shopping_money = float(input())
```

Но във втората част има детайл, с който трябва да се съобразим. Условието гласи следното:

Всеки следващ ред ще има определена команда. Когато получите командата "**mall.Enter**", на всеки следващ ред ще получите стрингове, съдържащи информация относно покупките/действията, които Лина иска да направи.

Тук е моментът, в който трябва да се съобразим, че от **втория ред нататък трябва да започнем да четем команди**, но **едва след като получим** команда **"mall.Enter"**, трябва да започнем да ги обработваме. Как можем да направим това? Използването на **while** цикъл е добър избор.

По-долу е дадено примерно решение как можем да пропуснем всички команди преди получаване на команда "mall.Enter".

Забележете, че извикването на `input()` след края на цикъла се използва за преминаване към първата команда за обработване.

```
command = input()
```

```
while command != "mall.Enter":  
    command = input()
```

```
    command = input()
```

## Алгоритъм за решаване на задачата

Алгоритъмът за решаването на самата задача е праволинеен - продължаваме да четем команди от конзолата, докато не бъде подадена команда "mall.Exit".

През това време обработваме всеки един ASCII знак (`char`) от всяка една команда спрямо правилата, указани в условието, и едновременно с това модифицираме парите, които Лина има, и съхраняваме броя на покупките.

Нека разгледаме първите два проблема пред нашия алгоритъм. Първият проблем засяга начина, по който можем да четем командите, докато не срещнем "mall.Exit". Решението, както видяхме по-горе, е да се използва `while` цикъл. Вторият проблем е задачата да достъпим всеки един знак от подадената команда. Имайки предвид, че входните данни с командите са от тип `string`, то най-лесният начин да достъпим всеки знак в тях е чрез `for-in` цикъл.

Ето как би изглеждало използване на два такива цикъла:

```
while command != "mall.Exit":  
    for action in command:  
        pass  
  
    command = input()
```

Следващата част от алгоритъма ни е да обработим символите от командите, спрямо следните правила от условието:

- Ако символът е **главна буква**, Лина получава 50% намаление, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация ASCII символа.
- Ако символът е **малка буква**, Лина получава 70% намаление, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация ASCII символа.
- Ако символът е "%", Лина прави покупка, която намалява парите ѝ на половина.

- Ако символът е "()", Лина изтегля пари от дебитната си карта и добавя към наличните си средства 10 лева.
- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на ASCII символа от наличните ѝ средства.

Нека разгледаме проблемите от първото условие, които стоят пред нас. Единият е как можем да разберем дали даден **символ представлява главна буква**. Можем да използваме един от двата начина:

- Имайки предвид, факта, че буквите в азбуката имат ред, можем да използваме проверката **action >= 'A' && action <= 'Z'**, за да проверим дали нашият символ се намира в интервала от големи букви.
- Можем да използваме функцията **.isupper(...)**.

Другият проблем е как можем **да пропуснем даден символ**, ако той представлява операция, която изисква повече пари, отколкото Лина има? Това е възможно да бъде направено чрез използване на **continue** конструкцията.

Примерната проверка за първата част от условието изглежда по следния начин:

```
if action >= 'A' and action <= 'Z':
    price = ord(action) * 0.5
    if shopping_money < price:
        continue

    shopping_money -= price
    purchases += 1
```

**Забележка:** **purchases** е променлива от тип **int**, в която държим броя на всички покупки.

Смятаме, че читателят не би трябвало да изпита проблем при имплементацията на всички други проверки, защото са много сходни с първата.

## Форматиране на изхода

В края на задачата трябва да **принтираме** определен **изход**, в зависимост от следното условие:

- Ако не са били направени никакви покупки – "No purchases. Money left: {останали пари} lv."
- Ако е направена поне една покупка - "{брой покупки} purchases. Money left: {останали пари} lv."

Операциите по принтиране са тривиални, като единственото нещо, с което трябва да се съобразим е, че **парите трябва да се принтират с точност от 2 символа след десетичния знак**.

Как можем да направим това? Ще оставим отговора на този въпрос на читателя.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1062#0>.

## Задача: числен израз

Бони е изключително могъща вещица. Тъй като силата на природата не е достатъчна, за да се бори успешно с вампири и върколаци, тя започнала да усвоява силата на Изразите. Изразът е много труден за усвояване, тъй като заклинанието разчита на способността за **бързо решаване на математически изрази**.

За използване на "Израз заклинание", вещицата трябва да знае резултата от математически израз предварително. **Израз заклинанието** се състои от няколко прости математически израза. Всеки математически израз може да съдържа оператори за **събиране, изваждане, умножение и/или деление**.

Изразът се решава **без да се вземат под внимание математическите правила** при пресмятане на числови изрази. Това означава, че приоритет има последователността на операторите, а не това какъв вид изчисление правят. Израза **може да съдържа скоби**, като **всичко в скобите се пресмята първо**. Всеки израз може да съдържа множество скоби, но не може да съдържа вложени скоби:

- Израз съдържащ (...(...)...)
- Израз съдържащ (...)...(...)

## Пример

Изразът

**4 + 6 / 5 + (4 \* 9 - 8) / 7 \* 2**

бива решен по следния начин:

```
4 + 6 / 5 + (4 * 9 - 8) / 7 * 2 =
10 / 5 + (4 * 9 - 8) / 7 * 2 =
2 + (4 * 9 - 8) / 7 * 2 =
2 + (36 - 8) / 7 * 2 =
2 + 28 / 7 * 2 =
30 / 7 * 2 =
4.285714285714286 * 2 =
8.57172857142571 =
8.57
```

Бони е много красива, но не чак толкова съобразителна, затова тя има нужда от нашата помощ, за да усвои силата на Изразите.

## Входни данни

Входните данни се състоят от един ред, който бива подаван от конзолата. Той съдържа **математическият израз за пресмятане**. Редът винаги завършва със символа " $=$ ". Символът " $=$ " означава **край на математическия израз**.

Входните данни винаги са валидни и във формата, който е описан. Няма нужда да бъдат валидириани.

## Изходни данни

Изходните данни трябва да се принтират на конзолата. Изходът се състои от един ред – резултата от **пресметнатия математически израз**.

Резултатът трябва да бъде закръглен до втората цифра след десетичния знак.

## Ограничения

- Изразите ще състоят от **максимум 2500 символа**.
- Числата от всеки математически израз ще са в интервала **[1 ... 9]**.
- Операторите в математическите изрази винаги ще бъдат измежду **+** (събиране), **-** (изваждане), **/** (деление) или **\*** (умножение).
- Резултатът от математическия израз ще е в интервала **[-100000.00 ... 100000.00]**.
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

## Примерен вход и изход

Вход	Изход
$4+6/5+(4*9-8)/7*2=$	8.57

Вход	Изход
$3+(6/5)+(2*3/7)*7/2*(9/4+4*1)=$	110.63

## Насоки и подсказки

Както обикновено, първо ще прочетем и обработим входа, след това ще решим задачата и накрая ще отпечатаме резултата, форматиран, както се изисква в условието.

## Обработване на входа

Входните данни се състоят от точно един ред от конзолата. За да имаме достъп до всеки един елемент от числовия израз, ще прочетем входа, чрез **input()** функцията и после ще превърнем входния **string** в масив от символи. Използвайки функцията **pop(...)** можем да достъпваме символите един по един:

```
expression = input()
symbols = list(expression)
symbol = symbols.pop(0)
```

### Алгоритъм за решаване на задачата

За целите на нашата задача ще имаме нужда от две помощни променливи:

- Една променлива, в която ще пазим **текущия резултат**.
- Още една променлива, в която ще пазим **текущия оператор** от израза.

```
result = 0
expression_operator = "+"
```

Относно кода по-горе трябва да поясним, че стойността по подразбиране на оператора е **+**, за да може още първото срещнато число да бъде събрано с резултата ни.

След като вече имаме началните променливи, трябва да помислим върху това, каква ще е основната структура на нашата програма. От условието разбираме, че **всеки израз завършва с =**, т.е. ще трябва да четем и обработваме символи, докато не срещнем **=**. Следва точното изписване на **while цикъл**:

```
while symbol != "=":
    symbol = symbols.pop(0)
```

Следващата стъпка е обработването на нашата **symbol** променлива. За нея имаме 3 възможни случая:

- Ако символът е **начало на подизраз, заграден в скоби**, т.е. срещнатият символ е **(**.
- Ако символът е **цифра между 0 и 9**. Но как можем да проверим това? Как можем да проверим дали символът ни е цифра? Тук идва на помощ **ASCII кодът на символа**, чрез който можем да използваме следната формула: **[ASCII кода на нашия символ] - [ASCII кода на символа 0] = [цифрата, която репрезентира символа]**. Ако резултатът от тази проверка е между 0 и 9, то тогава нашият символ наистина е **число**.

**Забележка:** за да вземем ASCII кода на даден символ, ще ползваме **ord(...)** функцията.

- Ако символът е **оператор**, т.е. е **+, -, \*** или **/**.

```
while symbol != "=":
    if symbol == "(":
        # TODO

    elif 0 <= ord(symbol) - ord('0') \
        and ord(symbol) - ord('0') <= 9:
        # TODO
```

```

elif (symbol == '+' or
      symbol == '-' or
      symbol == '/' or
      symbol == '*'):

# TODO

```

Нека разгледаме действията, които трябва да извършим при съответните случаи, които дефинирахме:

- Ако нашият символ е **оператор**, то тогава единственото, което трябва да направим, е да зададем нова стойност на променливата **expression\_operator**.
- Ако нашият символ е цифра, тогава трябва да променим текущия резултат от израза в зависимост от текущия оператор, т.е. ако **expression\_operator** е `-`, тогава трябва да намалим резултата с цифровата репрезентация на текущия символ. Можем да вземем цифровата репрезентация на текущия символ, чрез формулата, която използвахме при проверката на този случай **[ASCII кода на нашия символ] - [ASCII кода на символа 0] = [цифрата, която репрезентира символа]**.

```

elif 0 <= ord(symbol) - ord('0') \
      and ord(symbol) - ord('0') <= 9:
    if expression_operator == "+":
        result += ord(symbol) - ord('0')
    elif expression_operator == "-":
        result -= ord(symbol) - ord('0')
    elif expression_operator == "*":
        result *= ord(symbol) - ord('0')
    elif expression_operator == "/":
        result /= ord(symbol) - ord('0')

elif (symbol == '+' or
      symbol == '-' or
      symbol == '/' or
      symbol == '*'):
    expression_operator = symbol

```

- Ако нашият символ е `(`, това индикира началото на подизраз (израз в скоби). По дефиниция подизразът трябва да се калкулира преди да се модифицира резултата от целия израз (действията в скобите се извършват първи). Това означава, че ще имаме локален резултат за подизраза и локален оператор.

```

if symbol == "(":
    inner_result = 0
    inner_operator = '+'
    symbol = symbols.pop(0)

# TODO: Calculate the sub-expression

```

След това, за пресмятане стойността на подизраза използваме същите методи, които използвахме за пресмятане на главния израз - използваме **while цикъл**, за да четем символи (докато не срещнем символ `)`). В зависимост от това дали прочетения символ е цифра или оператор, модифицираме резултата на подизраза. Имплементацията на тези операции е аналогична с имплементацията за пресмятане на изрази, описана по-горе, затова смятаме, че читателят не би трябвало да има проблем с нея.

След като приключим калкулацията на резултата от подизраза ни, **модифицираме резултата на целия израз** в зависимост от стойността на **expression\_operator**:

```

if expression_operator == "+":
    result += ord(symbol) - ord('0')
elif expression_operator == "-":
    result -= ord(symbol) - ord('0')
elif expression_operator == "*":
    result *= ord(symbol) - ord('0')
elif expression_operator == "/":
    result /= ord(symbol) - ord('0')

```

## Форматиране на изхода

Единствения изход, който програмата трябва да принтира на конзолата, е **результатът от решаването на израза, с точност два символа след десетичния знак**. Как можем да форматираме изхода по този начин? Отговора на този въпрос оставяме на читателя.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1062#1>.

## Задача: бикове и крави

Всички знаем играта „Бикове и крави“ ([http://en.wikipedia.org/wiki/Bulls\\_and\\_cows](http://en.wikipedia.org/wiki/Bulls_and_cows)). При дадено 4-цифreno **тайно** число и 4-цифreno **предполагаемо** число, използваме следните правила:

- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число и е на **същата позиция**, имаме **бик**.
- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число, но е на **различна позиция**, имаме **крава**.

Ето няколко примера за бикове и крави:

Тайно число	1	4	8	1	Коментар
Предполагаемо число	8	8	1	1	Бикове = 1 Крави = 2
Предполагаемо число	9	9	2	4	Бикове = 0 Крави = 2

При дадено тайно число и брой на бикове и крави, нашата задача е **да намерим всички възможни предполагаеми числа** в нарастващ ред. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва да се отпечата "No".

## Входни данни

Входните данни се четат от конзолата. Входът се състои от 3 реда:

- Първият ред съдържа **секретното число**.
- Вторият ред съдържа **броя бикове**.
- Третият ред съдържа **броя крави**.

Входните данни ще бъдат винаги валидни. Няма нужда да бъдат проверявани.

## Изходни данни

Изходните данни трябва да се принтират на конзолата. Изходът трябва да се състои от **един единствен ред** – **всички предполагаеми числа**, разделени с единично празно място. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва **да се изпише "No"**.

## Ограничения

- Тайното число винаги ще се състои от **4 цифри в интервала [1 ... 9]**.
- Броят на **кравите и биковете** винаги ще е в интервала **[0 ... 9]**.
- Позволено време за изпълнение: **0.15 секунди**.
- Позволена памет: **16 MB**.

## Примерен вход и изход

Вход	Изход
2228 2 1	1222 2122 2212 2232 2242 2252 2262 2272 2281 2283 2284 2285 2286 2287 2289 2292 2322 2422 2522 2622 2722 2821 2823 2824

Вход	Изход
	2825 2826 2827 2829 2922 3222 4222 5222 6222 7222 8221 8223 8224 8225 8226 8227 8229 9222

Вход	Изход
1234	1134 1214 1224 1231 1232 1233 1235 1236 1237 1238 1239 1244
3	1254 1264 1274 1284 1294 1334 1434 1534 1634 1734 1834 1934
0	2234 3234 4234 5234 6234 7234 8234 9234

## Насоки и подсказки

Ще решим задачата на няколко стъпки:

- Ще прочетем **входните данни**.
- Ще генерираме всички възможни **четирицифренi комбинации** (кандидати за проверка).
- За всяка генерирана комбинация ще изчислим **колко бика и колко крави** има в нея спрямо секретното число. При съвпадение с търсените бикове и крави, ще **отпечатаме комбинацията**.

## Обработване на входа

За входа на нашата задача имаме 3 реда:

- Секретното число.
- Броят желани бикове.
- Броят желани крави.

Прочитането на тези входни данни е тривиално:

```
guess_number = input()
target_bulls = int(input())
target_cows = int(input())
```

## Алгоритъм за решаване на задачата

Преди да започнем писането на алгоритъма за решаване на нашия проблем, трябва да **декларираме флаг**, който да указва дали е намерено решение:

```
solution_found = False
```

Ако след приключването на нашия алгоритъм, този флаг все още е **False**, тогава ще принтираме **No** на конзолата, както е указано в условието:

```
if not solution_found:
    print("No")
```

Нека започнем да размишляваме над нашия проблем. Това, което трябва да направим, е да **анализираме всички числа от 1111 до 9999** без тези, които съдържат в себе си нули (напр. **9011, 3401** и т.н. са невалидни числа). Какъв е най-лесният начин за **генериране** на всички тези **числа?** С **вложени цикъла.** Тъй като имаме **4-цифрен** число, ще имаме **4 вложени цикъла**, като всеки един от тях ще генерира **отделна цифра** от нашето число за тестване:

```
for digit_1 in range(1, 10):
    for digit_2 in range(1, 10):
        for digit_3 in range(1, 10):
            for digit_4 in range(1, 10):
```

Благодарение на тези цикли, **имаме достъп до всяка една цифра** на всички числа, които трябва да проверим. Следващата ни стъпка е да **разделим секретното число на цифри.** Това може да се постигне много лесно чрез **разрязване на стринга** (*string slicing*). Алтернативно също можем да постигнем аналогичен резултат чрез достъпване на отделните символи по индекс:

```
guess_digit_1 = int(guess_number[0:1])
guess_digit_2 = int(guess_number[1:2])
guess_digit_3 = int(guess_number[2:3])
guess_digit_4 = int(guess_number[3:4])
```

Остават ни последните две стъпки преди да започнем да анализираме колко крави и бикове има в дадено число. Съответно, първата е **декларацията на counter** (**брояч**) **променливи** във вложените ни цикли, за да **броим кравите и биковете** за текущото число. Втората стъпка е да направим **копия на цифрите на текущото число**, което ще анализираме, за да предотвратим евентуални проблеми с работата на вложите цикли (напр., ако правим промени по цифрите):

```
digit_to_check_1 = digit_1
digit_to_check_2 = digit_2
digit_to_check_3 = digit_3
digit_to_check_4 = digit_4

current_bulls = 0
current_cows = 0
```

Вече сме готови да започнем анализирането на генерираните числа. Каква логика можем да използваме? Най-елементарният начин да проверим колко крави и бикове има в едно число е чрез **поредица от if-elif проверки.** Да, не е най-оптималният начин, но с цел да не използваме знания извън пределите на тази книга, ще изберем този подход.

От какви **проверки** имаме нужда?

Проверката за **бикове** е елементарна - проверяваме дали **първата цифра** от генерираното число е еднаква със **същата цифра** от секретното число. Премахваме проверените цифри с цел да избегнем повторения на бикове и крави:

```
# Find all bulls, count them and remove them (assign -1 and -2)
if digit_to_check_1 == guess_digit_1:
    # Bull at position #1 found -> count it and remove it
    current_bulls += 1
    guess_digit_1 = -1
    digit_to_check_1 = -2
```

Повтаряме действието за втората, третата и четвърта цифра.

Проверката за **крави** можем да направи по следния начин - първо проверяваме дали **първата цифра** от генерираното число **съвпада с втората, третата или четвъртата цифра** на секретното число. Примерна имплементация:

```
# Find all cows for digitToCheck1, count them and
# and remove them (assign -1)

if digit_to_check_1 == guess_digit_2:
    # Cow at position #2 found -> count it and remove it
    current_cows += 1
    guess_digit_2 = -1

elif digit_to_check_1 == guess_digit_3:
    # Cow at position #3 found -> count it and remove it
    current_cows += 1
    guess_digit_3 = -1

elif digit_to_check_1 == guess_digit_4:
    # Cow at position #4 found -> count it and remove it
    current_cows += 1
    guess_digit_4 = -1

# TODO: Check other digits
```

След това последователно проверяваме дали **втората цифра** от генерираното число **съвпада с първата, третата или четвъртата цифра** на секретното число, дали **третата цифра** от генерираното число съвпада с **първата, втората или четвъртата цифра** на секретното число и накрая проверяваме дали **четвъртата цифра** от генерираното число съвпада с **първата, втората или третата цифра** на секретното число.

## Отпечатване на изхода

След като приключим всички проверки, ни остава единствено да **проверим дали биковете и кравите в текущото генерирано число съвпадат с желаните бикове и крави, прочетени от конзолата**. Ако това е така, принтираме текущото число на конзолата:

```
if current_bulls == target_bulls \
    and current_cows == target_cows:
    if solution_found:
        print(" ", end="")
    print("%d%d%d%d" % (digit_1, digit_2,
                          digit_3, digit_4), end="")
solution_found = True
```

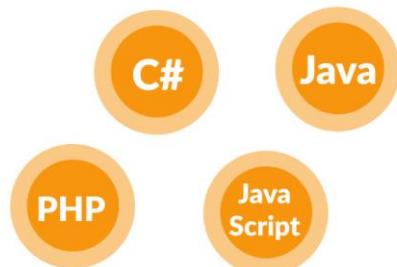
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1062#2>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтуни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтуни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 10. ФУНКЦИИ

В настоящата глава ще се запознаем с **функциите** и ще научим какво **представляват** те, както и кои са **базовите концепции** при работа с тях. Ще научим защо е **добра практика** да ги използваме, как да ги **дефинираме и извикваме**. Ще се запознаем с **параметри и връщана стойност от функция**, както и как да използваме тази връщана стойност. Накрая на главата, ще разгледаме **утвърдените практики** при използването на функции.

## Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=lDqjZuDcfbE>.

## Какво е функция?

До момента установихме, че при **писане** на програма, която решава даден проблем, ни **улесява** това, че **разделяме** задачата на **части**. Всяка част отговаря за **дадено действие** и по този начин не само ни е **по-лесно** да решим задачата, но и значително се подобрява както **четимостта** на кода, така и проследяването на логиката на програмата и намирането на грешки.

В контекста на програмирането, **функция** (метод) се нарича **именувана група от инструкции**, които изпълняват дадена функционалност. Тази група от инструкции е логически отделена и именувана, така че изпълнението на инструкциите в групата може да бъде стартирано чрез това име в хода на изпълнението на програмата. Стартоването на изпълнението на инструкциите във функцията се нарича **извикване на функцията** (на английски function call или invoking a function).

Една функция може да бъде извикана толкова пъти, колкото ние преценим, че ни е нужно за решаване на даден проблем. Това ни **спестява** повторението на един и същи код няколко пъти, което от своя страна **намалява** възможността да пропуснем грешка при евентуална корекция на въпросния код.

Ще разгледаме два типа функции - "**прости**" (без параметри) и "**сложни**" (с параметри).



В обектно-ориентираното програмиране (което не е предмет на тази книга) функциите, които са част от класове, се наричат **методи**. В някои езици за програмиране функциите се наричат още **процедури**.

## Прости функции

**Простите** функции отговарят за изпълнението на дадено **действие**, което **спомага** за решаване на определен проблем. Такива действия могат да бъдат например разпечатване на даден низ в конзолата, извършване на някаква проверка, изпълнение на цикъл и други.

Нека разгледаме следния **пример за проста функция**:

```
def print_header():
    print("-----")
```

Тази функция има задачата да отпечата заглавие, което представлява поредица от символа -. Поради тази причина името ѝ е **print\_header**. Кръглите скоби ( и ) винаги следват името на функцията, независимо как сме я именували. Важно е името на функциите, с които работим, да описва действието, което извършват. По-късно в тази глава ще разгледаме още утвърдени практики за избиране на имена на функциите.

**Тялото** на функцията съдържа **програмния код** (инструкциите), което се намира на следващия ред, след двоеточието и е изписано с една табулация навътре (с индентация). Двоеточието **винаги** следва **декларацията** ѝ и след него поставяме кода, който решава проблема, описан от името на функцията. Тялото на функцията се изписва по-навътре, обикновено 4 интервала (една табулация), които го обособяват като отделен блок инструкции, прилежащи към функцията.

Изпълнението на тази програма само по себе си няма да отпечата нищо на екрана, тъй като още не сме извикали функцията.

## Защо да използваме функции?

До тук установихме, че функциите спомагат за **разделянето на обемна задача на по-малки части**, което води до **по-лесно решаване** на въпросното задание. Това прави програмата ни не само по-добре структурирана и лесно четима, но и по-разбираема.

Чрез функциите **избягваме повторението** на програмен код. **Повтарящият** се код е **лоша практика**, тъй като силно **затруднява поддръжката** на програмата и води до грешки. Ако дадена част от кода ни присъства в програмата няколко пъти и се наложи да променим нещо, то промените трябва да бъдат направени във всяко едно повторение на въпросния код. Вероятността да пропуснем място, на което трябва да нанесем корекция, е много голяма, което би довело до некоректно поведение на програмата. Това е причината, поради която е **добра практика**, ако използваме даден фрагмент код **повече от веднъж** в програмата си, да го **дефинираме като отделна функция**.

Функциите ни предоставят **възможността** да използваме даден **код няколко пъти**. С решаването на все повече и повече задачи ще установите, че използването на вече съществуващи функции спестява много време и усилия.

## Дефиниране на функции

**Дефиниране на функция** представлява регистрирането на функцията в програмата, за да бъде разпознавана и да може да бъде използвана в останалата част от нея.

Със следващия пример ще разгледаме елементите в дефиницията на една функция в езика Python:

```
def calculate_square(num):
    return num * num
```

- **def.** Ключовата дума **def** в езика за програмиране Python показва, че желаем да дефинираме нова функция.
- **Име на функцията.** Името на функцията е **определеното от нас** и следва ключовата дума **def**, като не забравяме, че то трябва да **описва действието**, което се изпълнява от инструкциите в тялото ѝ. В примера името е **calculate\_square**, което ни указва, че задачата на тази функция е да изчисли квадрата на някое число.
- **Списък с параметри.** Дефинира се между скобите **(** и **)**, които изписваме след името на функцията. Тук изброяваме поредицата от **параметри**, които функцията ще използва. Може да присъства **само един** параметър, **няколко такива** или да е **празен** списък. Ако няма параметри, то записваме само скобите **()**. В конкретния пример декларираме един параметър **num**.
- **Двоеточие.** След затварящата скоба слагаме двоеточие **:**, което оказва, че започва тялото на функцията.
- **Имплементация (тяло).** В тялото на функцията описваме **алгоритъма** (инструкциите), по който тя решава даден проблем, т.е. тялото съдържа кода, който реализира **логиката** на функцията. В показания пример изчисляваме квадрата на дадено число, а именно **num \* num**. Тялото се записва на нов ред с индентация.

При дефиниране на функции е важно да спазваме тази конкретна **последователност** на елементите.

Когато дефинираме дадена променлива в тялото на една функция, я наричаме **локална** променлива за функцията. Областта, в която съществува и може да бъде използвана тази променлива, започва от реда, на който сме я дефинирали и стига до последната инструкция от блока на тялото (която се намира по-навътре). Тази област се нарича **област на видимост** на променливата (variable scope).



Някои езици за програмиране (например С или С++) различават **декларирането** и **дефинирането** на функции. **Декларирането** на функция информира компилатора или интерпретатора, че функцията със съответното име и параметри съществува, без да съдържа имплементация. **Дефиницията** съдържа имплементацията (тялото) на функцията. В езика **Python** това различаване не съществува и функциите винаги биват дефинирани, т.е. при създаването на функция винаги трябва да предоставим нейната имплементация.

## Извикване на функции

Извикването на функция представлява **стартирането** на **изпълнението** на кода, който се намира в **тялото на функцията**. Това става като изпишем **името** на

функцията, последвано от кръглите скоби **( )**. Ако функцията ни изисква входни данни (параметри), то те се подават в скобите **( )**, като последователността на подадените параметри трябва да съвпада с последователността на параметрите при дефинирането на функцията. Ето един пример:

```
# define function
def print_header():
    print("-----")

# invoke function
print_header()
```

Дадена функция може да бъде извикана от **няколко места** в нашата програма и повече от един път. Важно е да знаем, че в езика **Python**, ако една функция е дефинирана някъде в програмата, то тя може да бъде извиквана само след самата ѝ дефиниция.

Тъй като извикването на функция е инструкция сама по себе си, то можем безпроблемно от тялото на една функция да извикаме друга функция:

```
# define function
def print_header():
    # invoking function from
    # another function
    print_header_top()
    print_header_bottom()
```

Съществува вариант функцията да бъде извикана от **собственото си тяло**. Това се нарича **рекурсия** и можете да намерите повече информация за нея в [Wikipedia](#) или да потърсите сами в Интернет.

## Пример: празна касова бележка

Да се напише функция, който печата празна касова бележка. Функцията трябва да извиква други три функции: една за принтиране на заглавието, една за основната част на бележката и една за долната част.

Част от касовата бележка	Текст
Горна част	CASH RECEIPT -----
Средна част	Charged to_____ Received by_____
Долна част	----- (c) SoftUni

## Примерен вход и изход

Вход	Изход
(няма)	CASH RECEIPT ----- Charged to _____ Received by _____ ----- (c) SoftUni

## Насоки и подсказки

Първата ни стъпка е да създадем функция за **принтиране на заглавната част** от касовата бележка (header). Нека ѝ дадем смислено име, което описва кратко и ясно задачата ѝ, например **print\_receipt\_header**. В тялото ѝ ще напишем кода от примера по-долу:

```
def print_receipt_header():
    print("CASH RECEIPT")
    print("-----")
```

Съвсем аналогично ще създадем още две функции за разпечатване на средната част на бележката (body) **print\_receipt\_body** и за разпечатване на долната част на бележката (footer) **print\_receipt\_footer**.

След това ще създадем и **още една функция**, която ще извиква трите функции, които написахме до момента една след друга. Накрая ще **извикаме** функцията **print\_receipt** от нашата програма:

```
def print_receipt():
    print_receipt_header()
    print_receipt_body()
    print_receipt_footer()

print_receipt()
```

## Тестване в Judge системата

Програмата с общо четири функции, които се извикват една от друга, е готова и можем **да я изпълним и тестваме**, след което да я пратим за проверка в Judge системата: <https://judge.softuni.bg/Contests/Practice/Index/1063#0>.

## Функции с параметри (по-сложни функции)

Много често в практиката, за да бъде решен даден проблем, функцията, с чиято помощ постигаме това, се нуждае от **допълнителна информация**, която зависи от

задачата ѝ. Именно тази информация представляват параметрите на функцията и нейното поведение зависи от тях.

## Използване на параметри във функциите

Както отбелязахме по-горе, параметрите освен нула на брой, могат също така да са един или няколко. При декларацията им ги разделяме със запетая. Те могат да бъдат от различен тип данни (число, низ и т.н.), а по-долу е показан пример как точно те биват използвани в тялото на функцията.

Ето една примерна дефиниция на функция и списъка ѝ с параметри:

```
def print_numbers(start, end):
    for i in range(start, end):
        print(str(i))
```

В този пример имаме два параметъра, съответно с имената **start** и **end**. След дефиницията на функцията, можем да я използваме в програмата - **извикваме** функцията като ѝ **предаваме конкретни стойности** за параметрите:

```
print_numbers(5, 10)
```

С това извикване на функцията, тялото ѝ ще бъде изпълнено като за параметъра **start** ще бъде използвана стойността **5**, а за параметъра **end** - стойността **10**.

При **декларирането на параметри**, можем да използваме **различни** типове данни като всеки един параметър трябва да има **име** (което да е смислено). Важно е да отбележим, че при извикване на функцията, трябва да подаваме **стойности** за параметрите в **реда**, в който са **деклариирани**.

Нека разгледаме друга примерна дефиниция на функция, която има няколко параметъра от различен тип:

```
def print_student(name, age, grade):
    print("Student: " + name + ", age: "
          + str(age) + ", grade: " + str(grade))
```

## Пример: знак на цяло число

Да се създаде функция, която печата дали подаденото и цяло число е положително, отрицателно или нула.

### Примерен вход и изход

Вход	Изход
2	The number 2 is positive.
-5	The number -5 is negative.
0	The number 0 is zero.

## Насоки и подсказки

Първата ни стъпка е **създаването** на функция и даването ѝ на описателно име, например **print\_sign**. Тази функция ще има само един параметър - числото, чийто знак искаме да проверим:

```
def print_sign(n):
    # your code here
```

Следващата ни стъпка е да **имплементираме** логиката, чрез която ще се проверява дали подаденото число е положително, отрицателно или нула. От примерите виждаме, че има три случая - числото е по-голямо от нула, равно на нула или по-малко от нула, което означава, че ще направим **три проверки** в тялото на функцията.

Следващата ни стъпка е да прочетем входното число и да извикаме новата функция:

```
print_sign(input("Enter number: "))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#1>.

## Незадължителни параметри и подразбираща се стойност

Езикът Python позволява използването на **незадължителни параметри**. Това позволява **пропускането** на някой параметри при извикването на функцията. Декларирането им става чрез осигуряване на **стойност по подразбиране** в декларацията на съответния параметър.

Следният **пример** онагледява употребата на незадължителните параметри:

```
def print_numbers(start=0, end=100):
    for i in range(start, end):
        print(str(i))
```

Показаната функция **print\_numbers(...)** може да се извика по няколко начина:

```
print_numbers(5, 10)
print_numbers(15)
print_numbers()
print_numbers(end = 40, start = 35)
```

При първото извикване на функцията ще се използват параметрите **start = 5, end = 10**. Второто извикване - параметрите **start = 0, end = 15**. Третото извикване - параметрите **start = 0, end = 100**. А четвъртото извикване ще използва параметрите **start = 35** и **end = 45**.

## Пример: принтиране на триъгълник

Да се създаде функция, която принтира триъгълник, като в примерите.

## Примерен вход и изход

Вход	Изход	Вход	Изход
3	1 1 2 1 2 3 1 2 1	4	1 1 2 1 2 3 1 2 3 4 1 2 3 1 2 1

## Насоки и подсказки

Преди да създадем функция за принтиране на един ред с дадени начало и край, прочитаме входното число от конзолата. След това избираме съмислено име за функцията, което описва целта ѝ, например **print\_line**, и я имплементираме:

```
def print_line(start, end):
    for i in range(start, end + 1):
        print(str(i) + " ", end="")
    print("")
```

От задачите за рисуване на конзолата си спомняме, че е добра практика **да разделяме фигурата на няколко части**. За наше улеснение ще разделим триъгълника на три части - горна, средна и долната.

Следващата ни стъпка е с цикъл да разпечатаме **горната половина** от триъгълника:

```
for i in range(0, n):
    print_line(1, i)
```

След това разпечатваме **средната линия**:

```
print_line(1, n)
```

Накрая разпечатваме **долната част** от триъгълника, като този път стъпката на цикъла намалява:

```
for i in range(n - 1, 0, -1):
    print_line(1, i)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#2>.

## Пример: рисуване на запълнен квадрат

Да се нарисува на конзолата запълнен квадрат със страна N, като в примерите.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
4	<pre>----- - \ \ / \ - - \ \ / \ - -----</pre>	3	<pre>----- - \ \ / - - \ \ / - -----</pre>	2	<pre>-----</pre>

## Насоки и подсказки

Първата ни стъпка е да прочетем входа от конзолата. След това трябва да създадем функция, която ще принтира първия и последен ред, тъй като те са еднакви. Нека не забравяме, че трябва да ѝ дадем **описателно име** и да ѝ зададем като **параметър** дължината на страната:

```
def print_header_footer(n):
    for i in range(0, (n * 2)):
        print("-", end="")
    print("")
```

Следващата ни стъпка е да създадем функция, която ще рисува на конзолата средните редове. Отново задаваме описателно име, например `print_middle_row`:

```
def print_middle_row(n):
    print("-")
    for i in range(0, n - 1):
        print("\ \ /", end="")
    print("-")
```

Накрая извикваме създадените функции, за да нарисуваме целия квадрат:

```
side = int(input("Enter size:"))
print_header_footer(side)
# TODO: Draw the rest of the square
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#3>.

## Връщане на резултат от функции

До момента разглеждахме функции, които извършват дадено действие, например отпечатване на даден текст, число или фигура на конзолата. Освен този тип функции, съществуват и такива, които могат да **връщат** някакъв **резултат** от своето изпълнение - например резултатът от умножението на две числа. Именно тези функции ще разгледаме в следващите редове.

## Оператор return

За да върнем резултат от функция, използваме оператора **return**. Той трябва да бъде **използван в тялото** на функцията и указва на програмата да **спре изпълнението** си и да **върне** на извиквача на функцията определена **стойност**, която се определя от израза след въпросния оператор **return**. В примера по-долу имаме функция, която чете две имена от конзолата, съединява ги и ги връща като резултат:

```
def read_full_name():
    first_name = input("First name:")
    last_name = input("Last name:")
    return first_name + " " + last_name
```

Операторът **return** може да бъде използван и във функции, които не връщат резултат - след оператора не трябва да има израз. Изпълнението му води до това функцията да спре изпълнението си, без да връща никаква стойност. В този случай употребата на **return** е единствено за прекратяване на изпълнението на функцията. Възможно е и операторът **return** да бъде използван на повече от едно място в тялото на функцията.

В примера по-долу имаме функция, която сравнява две числа и връща резултат съответно **-1, 0** или **1** според това дали първият параметър е по-малък, равен или по-голям от втория параметър, подаден на функцията. Функцията използва ключовата дума **return** на три различни места, за да върне три различни стойности според логиката на сравненията на числата:

```
def compare_to(number1, number2):
    if number1 > number2:
        return 1
    elif number1 == number2:
        return 0
    else:
        return -1
```

Важно е да отбележим, че **результатът**, който се връща от функцията, може да бъде от **различен тип** - низ, цяло число, число с плаваща запетая и т.н.

### Кодът след **return** е недостъпен

След оператор **return** в дадена функция, изпълнението ѝ се прекратява и продължава на мястото, от където е извикана функцията. Ако след оператора **return** има други инструкции, то те няма да бъдат изпълнени. След **return** не трябва да има друг **return** в същата функция, защото вторият е недостъпен.

Някои редактори, в това число и **PyCharm**, ще ви информират за проблема чрез изрично предупреждение:

```
def foo(n):
    return n * n
    # the following instruction will not be executed
print("foo was executed.")
```

This code is unreachable more... (Ctrl+F1)



В програмирането не може да има два пъти оператор `return` един след друг, защото изпълнението на първия няма да позволи да се изпълни вторият. Понякога програмистите се шегуват с фразата **“ниши return; return; и да си ходим”**, за да обяснят, че логиката на програмата е объркана.

## Употреба на връщаната от функцията стойност

След като дадена функция е изпълнена и върне стойност, то тази стойност може да се използва по **няколко** начина. Първият е да присвоим резултата като стойност на променлива:

```
maximum = get_max(5, 10)
```

Вторият е резултатът да бъде използван в израз:

```
total = get_price() * quantity * 1.20
```

Третият е да подадем резултата от работата на функцията към **друга функция** чрез параметър:

```
age = int(input())
```

## Пример: пресмятане на лицето на триъгълник

Да се напише функция, която изчислява лицето на триъгълник по дадени основа и височина и връща стойността му.

### Примерен вход и изход

Вход	Изход
3	
4	6

### Насоки и подсказки

Първо създаваме функция, която да изчислява лицето на базата на две променливи - дължината на страната **a** и височината **h**:

```
def get_triangle_area(length, height):
    return (length * height) / 2
```

Следващата ни стъпка е да прочетем входните данни и да **извикаме новата функция** с тях. Резултатът **записваме в подходяща променлива** и извеждаме на екрана:

```
a = float(input("Enter length:"))
h = float(input("Enter height:"))
area = get_triangle_area(a, h)
print("Area: "+str(area))
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#4>.

## Пример: степен на число

Да се напише функция, която изчислява и връща резултата от повдигането на число на дадена степен.

### Примерен вход и изход

Вход	Изход
2	256
8	

Вход	Изход
3	256
4	

## Насоки и подсказки

Първата ни стъпка отново ще е да прочетем входните данни от конзолата. Следващата стъпка е да създадем функция, която ще приема два параметъра (числото и степента) и ще връща като резултат число от тип **float**:

```
def calculate_power(number, power):
    result = 0
    # TODO: calculate power
    # by using a loop (or the ** operator)
    return result
```

След като сме направили нужните изчисления, ни остава да извикаме дефинираната функция и да отпечатаме резултата.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#5>.

## ФУНКЦИИ, ВРЪЩАЩИ НЯКОЛКО СТОЙНОСТИ

До тук разглеждахме функции, които **не връщат стойности** и функции, които **връщат една единствена стойност**. В практиката често се срещат случаи, в които се нуждаем дадена функция да върне **повече от една стойност** като резултат.

За целта при използване на оператора `return` в езика Python отделяме стойностите, които искаме да върнем, със запетая. Следната функция приема за параметри две целочислени числа (**x** и **y**) и **връща две стойности** - резултата от целочислено деление на двете числа и остатъка от делението им:

```
def divide(x, y):
    result = x // y
    remainder = x % y
    return result, remainder

a, b = divide(19, 5)
```

Извикването на функцията става по същия начин както и за функциите, които не връщат стойности или връщат една единствена стойност. За да използваме стойностите върнати от функцията, то можем да присвоим резултатите на няколко променливи, отделени със запетая, както е показано в примера със **a** и **b**. След изпълнението на този пример, **a** ще съдържа стойността **3**, а **b** - стойността **4**.

## Варианти на функции

В много езици за програмиране една и съща функция може да е декларирана в **няколко варианта** с еднакво име и различни параметри. Това е известно с термина "function overloading" (или "method overloading"). Езикът за програмиране Python **не позволява** дефиниране на варианти на функции, но подобна функционалност може да бъде постигната с представените по-рано незадължителни параметри.

### Сигнатура на функцията

В програмирането **начинът**, по който се **идентифицира** една функция, е чрез **двойката елементи** от декларацията ѝ – **име** на функцията и **списък** от нейните параметри. Тези два елемента определят нейната **спецификация**, така наречена още **сигнатура** на функцията:

```
def print_name(name):
    print("Name: "+name)
```

В този пример сигнатурата на функцията е нейното име **print\_name**, както и нейният параметър **name**.

Ако в програмата ни има функции с **еднакви имена**, но с **различни параметри**, то казваме, че имаме **варианти на функции** (function или method overloading). Езикът за програмиране Python не позволява дефинирането на две функции с едно и също име.

## Варианти на функции в Python

Различни варианти за извикване на една функция в езика Python може да бъде постигнато чрез използването на **незадължителни параметри**, или по-конкретно - чрез предоставянето на стойности по подразбиране за дадени параметри. Нека

разгледаме примера от по-рано за функция с няколко незадължителни параметъра:

```
def print_numbers(start=0, end=100):
    for i in range(start, end):
        print(str(i))
```

Както вече видяхме, можем да извикваме тази функция по-различни начини, които наподобяват различни варианти на функцията:

```
print_numbers(5, 10)
print_numbers(15)
print_numbers()
print_numbers(end = 40, start = 35)
```

## Вложени функции (локални функции)

Нека разгледаме следния пример за функция, която изчислява лице на окръжност:

```
def circle_circumference(radius):
    pi = 3.14

    def circle_diameter(r):
        return r * 2

    return pi * circle_diameter(radius)
```

## Какво е локална функция?

Виждаме, че в този код, във функцията `circle_circumference(...)` има друга декларирана функция `circle_diameter(...)`. Тя се нарича **вложена функция** (nested function) или още - **локална функция**. Вложените функции могат да се декларират във всяка една функция и са видими и могат да бъдат извиквани само в тази функция. В примера по-горе функцията `circle_diameter(...)` може да бъде извикана само от тялото на функцията `circle_circumference(...)`, но не и извън него.

## Зашо да използваме локални функции?

С времето и практиката ще открием, че когато пишем код, често се нуждаем от функции, които бихме използвали **само един път**, или пък нужната ни функция става твърде дълга. По-нагоре споменахме, че когато една функция съдържа в себе си прекалено много редове код, то той става труден за поддръжка и четене. В тези случаи на помощ идват **вложените функции** - те предоставят възможност в дадена функция да се декларира друга функция, която ще бъде използвана например само веднъж. Това спомага кодът ни да е по-добре **подреден** и по-

лесно **четим**, което от своя страна спомага за по-бърза корекция на евентуална грешка в кода и намалява възможността за грешки при промени в програмната логика.

## Деклариране на вложени функции

Нека отново разгледаме примера от по-горе:

```
def circle_circumference(radius):
    pi = 3.14

    def circle_diameter(r):
        return r * 2

    return pi * circle_diameter(radius)
```

В този пример, функцията `circle_diameter(...)` е вложена функция, тъй като е декларирана в тялото на функцията `circle_circumference(...)`. Това означава, че функцията `circle_diameter(...)` може да бъде използвана само във функцията `circle_circumference(...)`, но не и извън нея. Ако се опитаме да извикваме функцията `circle_diameter(...)` извън функцията `circle_circumference(...)`, това ще доведе до грешка при изпълнението на програмата.

Вложените функции имат достъп до променливите, които се използват в съдържащата ги функция. В примера по-горе променливата `pi` може да бъде използвана от тялото на функцията `circle_diameter(...)`. Тази особеност на вложените функции ги прави много удобни помощници при решаването на дадена задача. Те спестяват време и код, които иначе бихме вложили, за да предаваме на вложените функции параметри и променливи, които се използват в функциите, в които са вложени.

## Утвърдени практики при работа с функции

В тази част ще се запознаем с някои **утвърдени практики** при работа с функции, свързани с именуването, подредбата на кода и неговата структура.

### Именуване на функции

Когато именуваме дадена функция е препоръчително да използваме **смислени имена**. Тъй като всяка функция **отговаря** за някаква част от нашия проблем, то при именуването ѝ трябва да вземем предвид **действието**, което тя **извършва**, т.е. добра практика е **името да описва нейната цел**.

В езика **Python** е прието имената на функциите да се изписват с **малки букви**, като отделните думи се отделят с **долната черта \_**. Добра практика е името на функцията да е съставено от глагол или от двойка: глагол и съществително име.

Няколко примера за **коректно** именуване на функции:

- `find_student`
- `load_report`
- `sine`

Няколко примера за **лошо** именуване на функции:

- `Method1`
- `DoSomething`
- `Handle_Stuff`
- `SampleMethod`
- `DIRTYHack`

Ако не можем да измислим подходящо име, то има голяма вероятност функцията да решава повече от една задача или да няма ясно дефинирана цел. В такива случаи е добре да помислим как да я разделим на няколко отделни функции.

## Именуване на параметрите на функциите

При именуването на **параметрите** на функции важат почти същите правила, както и при самите функции. Разликите тук са, че за имената на параметрите е добро да използваме съществително име или двойка от прилагателно и съществително име. Трябва да отбележим, че е добра практика името на параметъра да **указва** каква е **мерната единица**, която се използва при работа с него.

Няколко примера за **коректно** именуване на параметри:

- `first_name`
- `report`
- `speed_kmh`
- `users_list`
- `font_size_in_pixels`
- `font`

Няколко примера за **некоректно** именуване на параметри:

- `p`
- `p1`
- `p2`
- `populate`
- `LastName`
- `lastName`

## Още добри практики при работа с функции

Нека отново припомним, че една функция трябва да изпълнява **само една** точно определена **задача**. Ако това не може да бъде постигнато, то тогава трябва да помислим как да **разделим** функцията на няколко отделни такива. Както казахме, името на функцията трябва точно и ясно да описва нейната цел. Добра добра практика в програмирането е да **избягваме** функции, по-дълги от екрана ни (приблизително). Ако все пак кода стане много обемен, то е препоръчително функцията да се **раздели** на няколко по-кратки, както в следния пример:

```
def print_receipt():
    print_receipt_header()
    print_receipt_body()
    print_receipt_footer()

print_receipt()
```

## Структура и форматиране на кода

При писането на функции трябва да внимаваме да спазваме коректна **индентация** (отместване навътре с една табулация). В езикът Python грешната **индентация** много често автоматично води до грешна програмата или такава, която не може да бъде изпълнена.

Пример за **правилно** форматиран Python код:

```
def read_full_name():
    first_name = input("First name:")
    last_name = input("Last name:")
    return first_name + " " + last_name
```

Пример за **некоректно** форматиран Python код (затова и последният ред е подчертан в червено):

```
def read_full_name():
    first_name = input("First name:")
last_name = input("Last name:")
    return first_name + " " + last_name
```

Друга добра практика при писане на код е да **оставяме празен ред** след циклите и условните конструкции и **два празни реда** след дефиницията на функции. Също така, опитвайте да **избягвате** да пишете **дълги редове и сложни изрази**. С времето ще установите, че това подобрява четимостта на кода и спестява време.

## Какво научихме от тази глава?

В тази глава се запознахме с базовите концепции при работа с функции:

- Научихме, че **целта** на функциите е да **разделят** големи програми с много редове код на по-малки и ясно обособени задачи.
- Запознахме се със **структурата** на функциите, как да ги **декларираме** и **извикваме** по тяхното име.
- Разгледахме примери за функции с **параметри** и как да ги използваме в нашата програма.
- Научихме какво представляват **сигнатурата** и **връщаната стойност** на функцията, както и какво представлява операторът **return**.
- Запознахме се с **добрите практики** при работа с функции - как да именуваме функциите и техните параметри, как да форматираме кода и други.

## Упражнения

За да затвърдим работата с функции, ще решим няколко задачи. В тях се изиска да напишете функция с определена функционалност и след това да я извикате като ѝ подадете данни, прочетени от конзолата, точно както е показано в примерния вход и изход.

### Задача: "Hello, Име!"

Да се напише функция, която получава като параметър име и принтира на конзолата "Hello, *име*!".

#### Примерен вход и изход

Вход	Изход
Peter	Hello, Peter!

#### Насоки и подсказки

Дефинираме функция **print\_name(name)** и я имплементираме, след което прочитаме от конзолата низ (името на човек) и извикваме функцията като ѝ подаваме прочетеното име.

#### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#7>.

### Задача: по-малко число

Да се създаде функция **get\_min(a, b)**, която връща по-малкото от две числа. Да се напише програма, която чете като входни данни от конзолата три числа и печата най-малкото от тях. Да се използва функцията **get\_min(...)**, която вече е създадена.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
1		-100	
2	1	-101	
3		-102	

## Насоки и подсказки

Дефинираме функция **get\_min(a, b)** и я имплементираме, след което я извикваме от програмата, както е показано по-долу. За да намерим минимума на три числа, намираме първо минимума на първите две от тях и след това минимума на резултата и третото число:

```
minimum = get_min(get_min(number1, number2), number3)
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#8>.

## Задача: повторяне на низ

Да се напише функция **repeat\_string(str, count)**, която получава като параметри променлива от тип низ и цяло число **N** и връща низа, повторен **N** пъти. След това резултатът да се отпечата на конзолата.

## Примерен вход и изход

Вход	Изход	Вход	Изход
str 2	strstr	roki 6	rokirokirokirokirokiroki

## Насоки и подсказки

Допишете функцията по-долу като добавите съединяването на входния низ към резултата в цикъла:

```
def repeat_string(str, count):
    repeated_string = ""
    for i in range(0, count):
        # TODO
    return repeated_string
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1063#9>.

## Задача: N-та цифра

Да се напише функция **`find_nth_digit(number, index)`**, която получава число и индекс **N** като параметри и печата N-тата цифра на числото (като се брои от дясно на ляво, започвайки от 1). След това, резултатът да се отпечата на конзолата.

### Примерен вход и изход

Вход	Изход
83746 2	4

Вход	Изход
93847837 6	8

Вход	Изход
2435 4	2

### Насоки и подсказки

За да изпълним алгоритъма, ще използваме **`while`** цикъл, докато дадено число не стане 0. На всяка итерация от **`while`** цикъла ще проверяваме дали настоящият индекс на цифрата не отговаря на индекса, който търсим. Ако отговаря, ще върнем като резултат цифрата на индекса (**`number % 10`**). Ако не отговаря, ще премахнем последната цифра на числото (**`number = number / 10`**). Трябва да следим коя цифра проверяваме по индекс (от дясно на ляво, започвайки от 1). Когато намерим цифрата, ще върнем индекса.

### Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1063#10>.

## Задача: число към бройна система

Да се напише функция **`integer_to_base(number, to_base)`**, която получава като параметри цяло число и основа на бройна система и връща входното число, конвертирано към посочената бройна система. След това, резултатът да се отпечата на конзолата. Входното число винаги ще е в бройна система 10, а параметърът за основа ще е между 2 и 10.

### Примерен вход и изход

Вход	Изход
3 2	11

Вход	Изход
4 4	10

Вход	Изход
9 7	12

### Насоки и подсказки

За да решим задачата, ще декларираме една променлива, в която ще пазим резултата. След това трябва да изпълним следните изчисления, нужни за конвертиране на числото:

- Изчисляваме **остатъка** от числото, разделено на основата.
- Вмъкваме **остатъка** от числото в началото на низа, представящ резултата.
- Разделяме числото на основата.
- Повтаряме алгоритъма, докато входното число не стане 0.

Допишете липсващата логика във функцията по-долу:

```
def integer_to_base(number, to_base):
    result = ""
    while number != 0:
        # implement the missing conversion logic

    return result
```

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1063#11>.

## Задача: известия

Да се напише програма, която прочита цяло число **N** и на следващите редове въвежда **N съобщения** (като за всяко съобщение се прочитат по няколко реда). За всяко съобщение може да се получат различен брой параметри. Всяко съобщение започва с **message\_type**: **success**, **warning** или **error**:

- Когато **message\_type** е **success** да се четат **operation + message** (всяко на отделен ред).
- Когато **message\_type** е **warning** да се чете само **message**.
- Когато **message\_type** е **error** да се четат **operation + message + errorCode** (всяко на отделен ред).

На конзолата да се отпечата **всяко прочетено съобщение**, форматирано в зависимост от неговия **message\_type**. Като след заглавния ред за всяко съобщение да се отпечатат толкова на брой символа **=**, **колкото е дълъг** съответният **заглавен ред** и да се сложи по един **празен ред** след всяко съобщение (за по-детайлно разбиране погледнете примерите).

Задачата да се реши с дефиниране на четири функции:

- **show\_success\_message()**
- **show\_warning\_message()**
- **show\_error\_message()**
- **read\_and\_process\_message()**

Ето как могат да изглеждат техните дефиниции:

```

def show_success_message(operation, message):
    # TODO

def show_warning_message(operation, message):
    # TODO

def show_error_message(operation, message, error_code):
    # TODO

def read_and_process_message():
    # TODO

```

### Примерен вход и изход

Вход	Изход
4 error credit card purchase Invalid customer address 500 warning Email not confirmed success user registration User registered successfully warning Customer has not email assigned	Error: Failed to execute credit card purchase. ===== Reason: Invalid customer address. Error code: 500.  Warning: Email not confirmed. =====  Successfully executed user registration. ===== User registered successfully.  Warning: Customer has not email assigned. =====

### Насоки и подсказки

Дефинираме и имплементираме посочените четири функции в условието.

В функцията **read\_and\_process\_message()** прочитаме типа съобщение от конзолата и според прочетения тип прочитаме останалите данни (който може да са още един два или три реда). След това извикваме съответния метод за печатане на съответния тип съобщение.

### Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1063#12>.

## Задача: числа към думи

Да се напише функция **letterize(number)**, която прочита цяло число и го разпечатва с думи на английски език според условията по-долу:

- Да се отпечатат с думи стотиците, десетиците и единиците (и евентуални минус) според правилата на английския език.
- Ако числото е по-голямо от **999**, трябва да се принтира "too large".
- Ако числото е по-малко от **-999**, трябва да се принтира "too small".
- Ако числото е **отрицателно**, трябва да се принтира "minus" преди него.
- Ако числото не е съставено от три цифри, не трябва да се принтира.

### Примерен вход и изход

Вход	Изход	Вход	Изход
3 999 -420 1020	nine-hundred and ninety nine minus four-hundred and twenty too large	2 15 350	fifteen three-hundred and fifty

Вход	Изход	Вход	Изход
4 311 418 509 -9945	three-hundred and eleven four-hundred and eighteen five-hundred and nine too small	3 500 123 9	five-hundred one-hundred and twenty three nine

### Насоки и подсказки

Можем първо да отпечатаме **стотиците** като текст - **(числото / 100) % 10**, след тях **десетиците** - **(числото / 10) % 10** и накрая **единиците** - **(числото % 10)**.

Първият специален случай е когато числото е точно **закръглено на 100** (напр. 100, 200, 300 и т.н.). В този случай отпечатваме "one-hundred", "two-hundred", "three-hundred" и т.н.

Вторият специален случай е когато числото, формирано от последните две цифри на входното число, е **по-малко от 10** (напр. 101, 305, 609 и т.н.). В този случай отпечатваме "one-hundred and one", "three-hundred and five", "six-hundred and nine" и т.н.

Третият специален случай е когато числото, формирано от последните две цифри на входното число, е **по-голямо от 10 и по-малко от 20** (напр. 111, 814, 919 и т.н.).

В този случай отпечатваме "one-hundred and eleven", "eight-hundred and fourteen", "nine-hundred and nineteen" и т.н.

## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1063#13>.

### Задача: криптиране на низ

Да се напише функция **encrypt(letter)**, който криптира дадена буква по следния начин:

- Вземат се първата и последна цифра от ASCII кода на буквата и се залепят една за друга в низ, който ще представя резултата.
- Към началото на стойността на низа, който представя резултата, се залепя символа, който отговаря на следното условие:
  - ASCII кода на буквата + последната цифра от ASCII кода на буквата.
- След това към края на стойността на низа, който представя резултата, се залепя символа, който отговаря на следното условие:
  - ASCII кода на буквата - първата цифра от ASCII кода на буквата.
- Функцията трябва да върне като резултат криптирания низ.

Пример:

- $j \rightarrow p16i$ 
  - ASCII кодът на **j** е **106**  $\rightarrow$  Първа цифра - **1**, последна цифра - **6**.
  - Залепяме първата и последната цифра  $\rightarrow$  **16**.
  - Към **началото** на стойността на низа, който представя резултата, залепяме символа, който се получава от събира на ASCII кода + последната цифра  $\rightarrow 106 + 6 \rightarrow 112 \rightarrow p$ .
  - Към **края** на стойността на низа, който представя резултата, залепяме символа, който се получава от разликата на ASCII кода - първата цифра  $\rightarrow 106 - 1 \rightarrow 105 \rightarrow i$ .

Използвайки функцията, описана по-горе, да се напише програма, която чете **поредица от символи, криптира ги** и отпечатва резултата на един ред.

Приемаме, че входните данни винаги ще бъдат валидни. От конзолата трябва да се прочетат входните данни, подадени от потребителя – цяло число **N**, следвани от по един символ на всеки от следващите **N** реда.

Да се криптират символите и да се добавят към криптирания низ. Накрая като резултат трябва да се отпечата **криптиран низ от символи** като в следващия пример:

- $S, o, f, t, U, n, i \rightarrow V83Kp11nh12ez16sZ85Mn10mn15h$

## Примерен вход и изход

Вход	Изход	Вход	Изход
4 s l a p	x15rt18kh97Xr12o	7 S o f t U n i	V83Kp11nh12ez16sZ85Mn10mn15h

## Насоки и подсказки

Създаваме една променлива **result**, в която ще се пази стойността на резултата, и ѝ присвояваме първоначална стойност **""** (празен низ). Трябва да се завърти цикъл **n** пъти, като на всяка итерация към променливата, в която пазим стойността на резултата, ще прибавяме криптириания символ.

За да намерим първата и последната цифри от ASCII кода, ще използваме алгоритъма, който използвахме за решаване на задача "Число към бройна система".

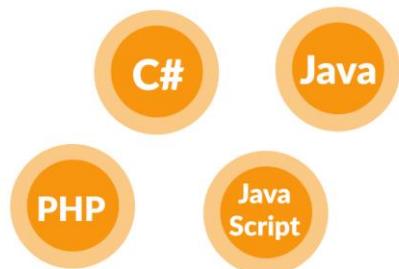
## Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/1063#14>.

Качествено образование,  
професия и работа за

## Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

**Софтууни работи пряко с компаниите от софтуерната индустрия**, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 11. Хитрости и хакове

В настоящата глава ще разгледаме някои хитрости, хакове и техники, които ще улеснят работата ни с езика **Python** в средата за разработка PyCharm. Поподробно ще се запознаем:

- Как правилно да **форматираме код**.
- С конвенции за **именуване на елементи от код**.
- С някои **бързи клавиши** (keyboard shortcuts).
- С някои **шаблони с код** (code snippets).
- С техники за **дебъгване на код**.

## Форматиране на кода

Правилното форматиране на нашия код ще го направи **по-четим и разбираем**, в случай че се наложи някой друг да работи с него. Това е важно, защото в практиката ще е необходимо да работим в екип с други хора и е от голямо значение дали пишем кода си така, че колегите ни да могат **бързо да се ориентират** в него.

Има определени правила за правилно форматиране на кода, които събрани в едно се наричат **конвенции**. Конвенциите са група от правила, общоприети от програмистите на даден език, и се ползват масово. Тези конвенции помагат за изграждането на норми в дадени езици – как е най-добре да се пише и какви са **добрите практики**. Приема се, че ако един програмист ги спазва, то кодът му е лесно четим и разбираем.

Езикът **Python** е създаден от организацията **Python Software Foundation** (PSF). Добрите практики за писане може да се различават между проектите, но общоприетите такива са дело на авторите на езика (основно Guido van Rossum). Трябва да знаете, че дори да не спазвате конвенциите, препоръчани от **PSF**, кодът ви ще работи (стига да е написан правилно), но просто няма да бъде лесно разбираем. Това, разбира се, не е фатално на основно ниво, но колкото по-бързо свикнете да пишете качествен код, толкова по-добре.

Официалната **Python код конвенция** на PSF е публикувана в страницата **PEP 8 – Style Guide for Python Code**: <https://www.python.org/dev/peps/pep-0008/>. Важно е да се отбележи, че в примерите, които сме давали до сега и ще даваме занапред в тази книга, се ръководим основно от нея.

За форматиране на кода от PSF се препоръчва всяко ниво на влагане да бъде **4 празни полета навътре** (или една табулация), както е в примера по-долу. Езикът не позволява смесването на табове и интервали:

```
if some_condition:  
    print("Inside the if statement")
```

Възможно е вместо 4, да се използват 2 интервала, но не се препоръчва.

Ето как изглеждат две вложени конструкции. Всеки блок започва с 4 интервала навътре. Броят интервали в началото на всеки ред определя какво е нивото му на влагане:

```
if some_condition:
    print("Start of outer if statement")
    if another_condition:
        print("Inside the inner if statement")
    print("End of outer if statement")
```

В **Python** форматирането на кода е от изключително важно значение за функционирането му. Код, който не е форматиран правилно, дава **грешка**.

Ето това е пример за **лошо форматиран код** спрямо общоприетите конвенции за писане на код на езика Python:

```
if some_condition:
    print("Inside the if statement")
```

Когато бъде стартиран, кодът дава следната грешка и програмата спира работа:

- **IndentationError: expected an indented block**

Командата вътре в **if** конструкцията трябва да бъде **4 празни полета навътре** (**един таб**). Веднага след ключовата дума **if** и преди условието на проверката се оставя **интервал**.

Същото правило важи и за **for** цикли и всякакви други вложени конструкции. Ето още един пример:

Правилно:

```
for i in range(5):
    print(i)
```

Грешно:

```
for i in range(5):
print( i )
```

За наше удобство има **бързи клавиши** в **PyCharm**, за които ще обясним по-късно в настоящата глава, но засега ни интересува една комбинация. Тя е за **форматиране на кода** в целия документ: **[CTRL + ALT + L]**.

Нека използваме **грешния пример** от преди малко:

```
for i in range(5):
print( i )
```

Ако натиснем [Ctrl + Alt + L], което е нашата комбинация за форматиране на **целия документ**, ще получим код, форматиран според **общоприетите конвенции за Python**, който ще изглежда по следния начин:

```
for i in range(5):
    print(i)
```

Тази комбинация може да ни помогне, ако попаднем на лошо форматиран код. Автоматичното форматиране обаче не влияе на именуването на нашите променливи (както и на други елементи на кода), за което ние трябва да се погрижим сами.

## Именуване на елементите на кода

В тази секция ще се фокусираме върху **общоприетите конвенции за именуване на проекти, файлове и променливи**, наложени от PSF.

### Именуване на проекти и файлове

За именуване на проекти и файлове се препоръчва описателно име, което подсказва **каква е ролята** на въпросния файл / проект и в същото време се препоръчва **lowercase\_with\_underscores** конвенцията. Това е **конвенция за именуване** на елементи, при която всяка дума, включително първата, започва с малка буква, а отделните думи са съединени с долни черти (\_), например **expression\_calculator**. Допустимо е да именуваме проекти и по конвенцията **PascalCase** – всяка дума, включително и първата, започва с главна буква, а отделните думи са долепени – **ExpressionCalculator**.

Пример: в този курс се започва с лекция на име **First steps in coding** и следователно едно примерно именуване на проекта за тази лекция може да бъде **first\_steps\_in\_coding** или **FirstStepsInCoding**. Файловете в даден проект задължително трябва да спазват конвенцията **lowercase\_with\_underscores**. Ако вземем за пример първата задача от лекцията **First steps in coding**, тя се казва **Hello World** и следователно нашият файл в проекта ще се казва **hello\_world.py**.

### Именуване на променливи

В програмирането променливите пазят някакви данни и за да е по-разбираем кода, името на една променлива трябва **да подсказва нейното предназначение**. Ето и още няколко препоръки за имената на променливите:

- Името трябва да е **кратко и описателно** и да обяснява за какво служи дадената променлива.
- Името трябва да се състои само от буквите **a-z, A-Z, цифрите 0-9**, както и **символа '\_'**.
- В Python е прието имената на променливите да спазват конвенцията **lowercase\_with\_underscores**.

- Трябва да се внимава за главни и малки букви, тъй като Python прави разлика между тях. Например **age** и **Age** са различни променливи.
- Имената на променливите **не могат да съвпадат със служебна дума** (keyword) от езика Python, например **for** е невалидно име на променлива.



Въпреки че използването на главни букви в имената на променливите е разрешено, **в Python това не се препоръчва** и се счита за лош стил на именуване.

Ето няколко примера за **добре именувани** променливи:

- **`first_name`**
- **`age`**
- **`start_index`**
- **`last_negative_number_index`**

Ето няколко примера за **лошо именувани променливи**, макар и имената да са коректни от гледна точка на интерпретатора на Python:

- **`firstName`** (именувана е по друга конвенция, неприета в Python).
- **`AGE`** (изписана е с главни букви).
- **`Start_Index`** (съдържа главни букви).
- **`lastNegativeNumber_Index`** (няма '\_' около всяка дума).

Първоначално всички тези правила може да ви се струват безсмислени и ненужни, но с течение на времето и натрупването на опит ще видите нуждата от норми за писане на качествен код, за да може да се работи по-лесно и по-бързо в екип. Ще разберете, че е изключително досадна работата с код, който е написан без да се спазват никакви правила за качествен код.

## Бързи клавиши в PyCharm

В предната секция споменахме за една от комбинациите, които се отнасят за форматиране на код: **[Ctrl + Alt + L]**, която беше за **форматиране на целия код в даден файл**. Тези комбинации се наричат **бързи клавиши** и сега ще дадем по-подробна информация за тях.

Бързи клавиши са **комбинации**, които ни предоставят възможността да извършваме някои действия **по-лесно и по-бързо**, като всяка среда за разработка на софтуер си има своите бързи клавиши, въпреки че повечето се повтарят. Сега ще разгледаме някои от **бързите клавиши** в PyCharm.

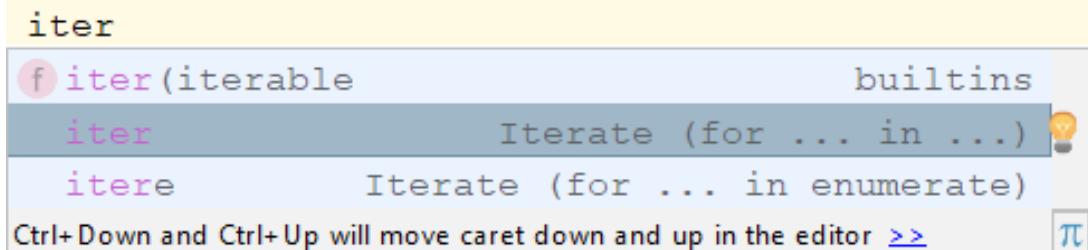
Комбинация	Действие
<b>[CTRL + F]</b>	Отваря търсачка, с която можем да <b>търсим в нашия код</b> .

Комбинация	Действие
[CTRL + /]	Закоментира част от кода. Разкоментира код, който е вече закоментиран.
[CTRL + Z]	Връща една промяна назад (т.нр. Undo).
[CTRL + SHIFT + Z]	Има противоположно действие на [CTRL + Z] (т.нр. Redo).
[CTRL + ALT + L]	Форматира кода, следвайки код конвенциите по подразбиране.
[CTRL + Backspace]	Изтрива думата вляво от курсора.
[CTRL + Del]	Изтрива думата вдясно от курсора.
[CTRL + S]	Запазва всички файлове в проекта.

Повече за **бързите клавиши** в PyCharm може да намерите в сайта на JetBrains: <https://jetbrains.com/help/pycharm/mastering-keyboard-shortcuts.html>.

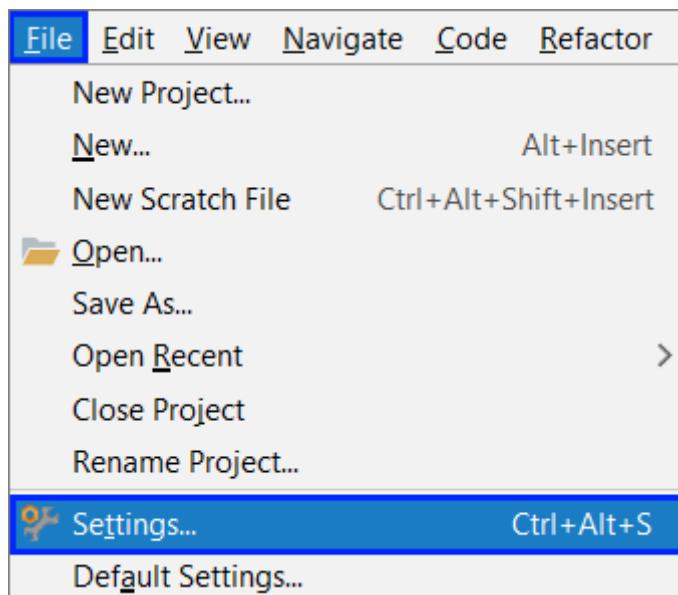
## Шаблони с код (code snippets)

В PyCharm съществуват т.нр. **шаблони с код** (live templates), при изписването на които се изписва по шаблон някакъв блок с код. Примерно, при изписването на кратък код "**iter**" и натискане на **[Tab]** се генерира кодът **for ... in ...:** в тялото на нашата програма, на мястото на краткия код. Това се нарича "разгъване на шаблон за кратък код". На фигурата по-долу е показано действието на шаблона "**iter**".

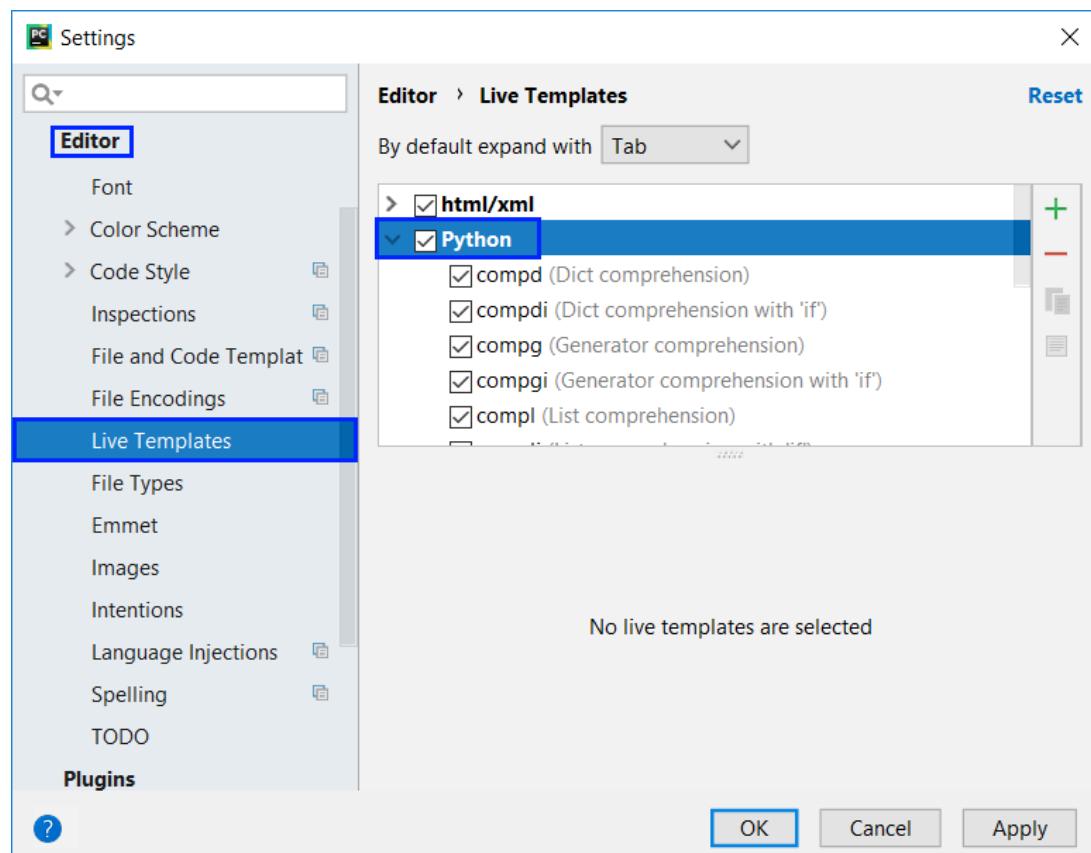


## Да си направим собствен шаблон за код

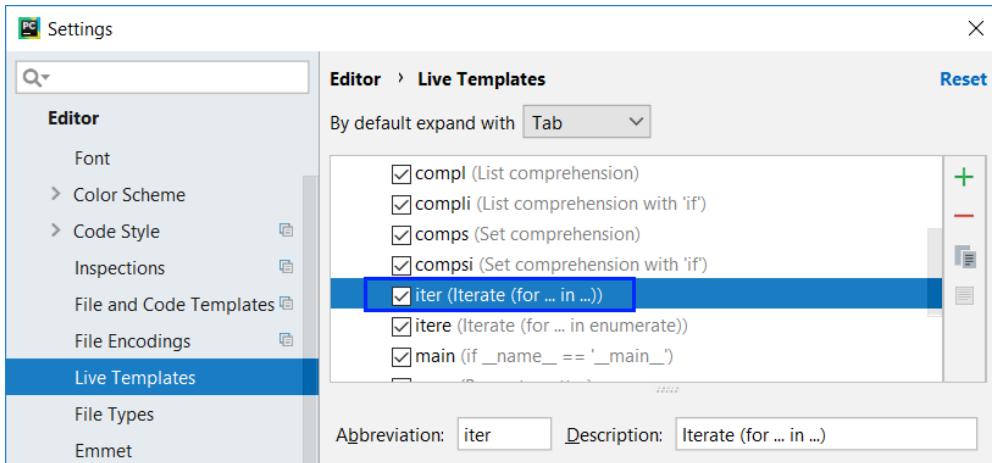
В тази секция ще покажем как сами да си **направим собствен шаблон**. Ще разгледаме **как се прави live template** за **два вложени цикъла**. Като начало ще си създадем нов празен проект и ще отидем на **[File] -> [Settings]**, както е показано на снимката:



В отворилия се прозорец трябва да изберем [Editor] -> [Live Templates], а от появилите се секции трябва да изберем стрелката **преди** отметката Python. Там се намират всички съществуващи шаблони за езика Python:

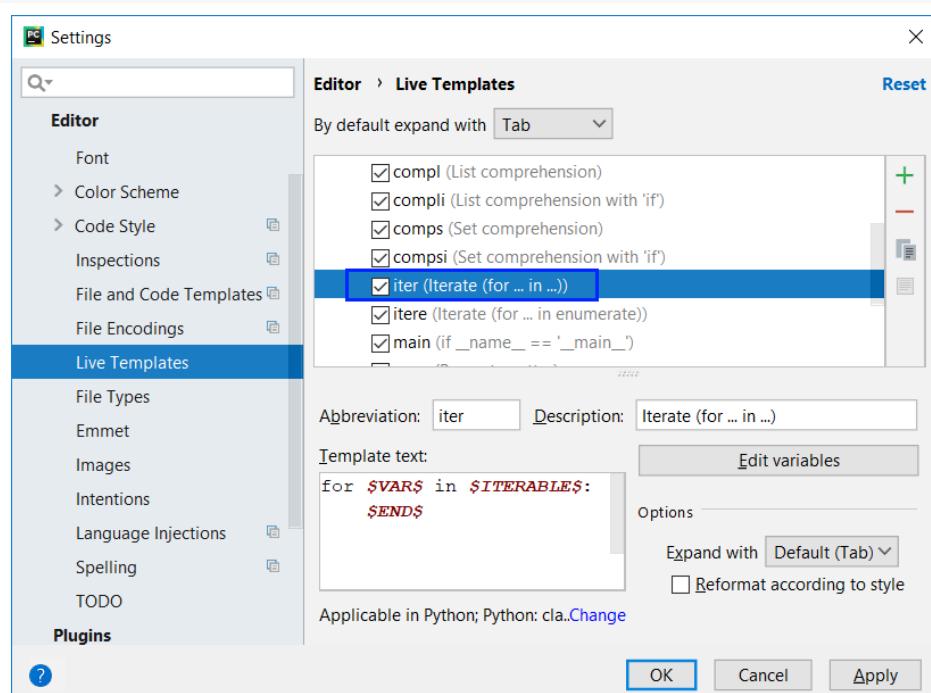


Избираме някой snippet, например **iter** и го разглеждаме:

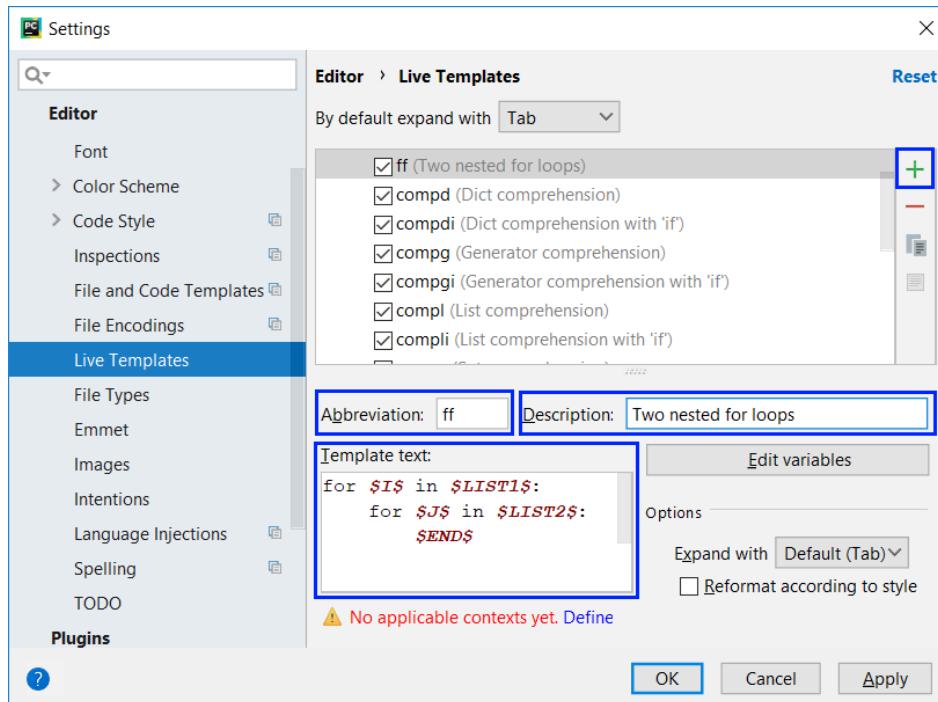


Виждаме доста непознати неща, но няма страшно, по-нататък ще се запознаем и с тях. Засега искаме да създадем собствен шаблон. За целта избираме бутона Add (зеленият + вдясно). От появилия се списък избираме [1. Live Template]. В долната част на прозореца, на **Abbreviation** пишем краткото име, с което ще извикваме шаблона (например **ff**, от **for**, **for**), за **Description** даваме полезно описание, а при **Template Text** въвеждаме следното:

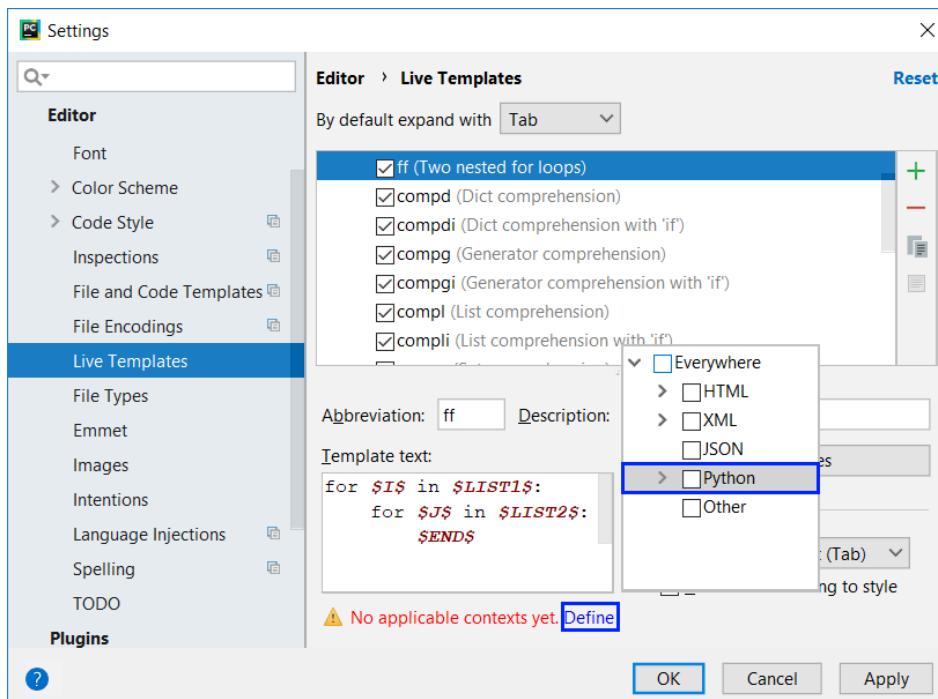
```
for $I$ in $LIST1$:
    for $J$ in $LIST2$:
        $END$
```



Получаваме предупреждение, че трябва да изберем контекст, т.е. в кои случаи да се показва нашият шаблон (червения текст под Template text). Избираме **Define** и от появилото се меню слагаме отметка пред **Python**:



Вече когато напишем **ff** в PyCharm, нашият нов live template се появява:



## Техники за дебъгване на кода

Дебъгването играе важна роля в процеса на създаване на софтуер, която ни позволява **постъпково да проследим изпълнението** на нашата програма. С помощта на тази техника можем да **следим стойностите на локалните променливи**, тъй като те се променят по време на изпълнение на програмата, и да **отстраним евентуални грешки** (бъгове). Процесът на дебъгване включва:

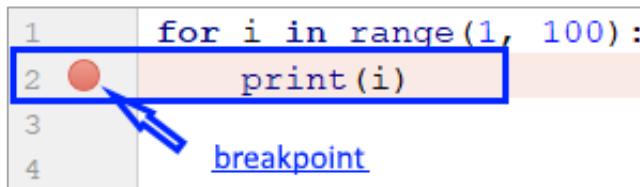
- **Забелязване** на проблемите (бъговете).
- **Намиране** на кода, който причинява проблемите.
- **Коригиране** на кода, причиняващ проблемите, така че програмата да работи правилно.
- **Тестване**, за да се убедим, че програмата работи правилно след нанесените корекции.

PyCharm ни предоставя **вграден дебъгер** (debugger), чрез който можем да поставяме **точки на прекъсване** (или breakpoints), на избрани от нас места. При среща на **стопер** (breakpoint), програмата **спира изпълнението** си и позволява **постъпково изпълнение** на останалите редове от кода. Дебъгването ни дава възможност да **вникнем в детайлите на програмата** и да видим къде точно възникват грешките и каква е причината за това.

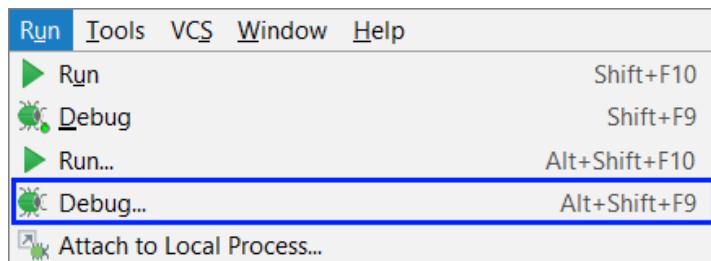
За да демонстрираме работа с дебъгера ще използваме следната програма:

```
for i in range(1, 100):
    print(i)
```

Ще сложим **стопер** (breakpoint) на функцията **print(...)**. За целта трябва да преместим курсора на реда, който печата на конзолата, и да натиснем **[CTRL+F8]**. Появява се **стопер**, където програмата ще **спре** изпълнението си:



За да стартираме програмата в режим на дебъгване, избираме [Run] -> [Debug ...] или натискаме **[Alt + Shift + F9]**:



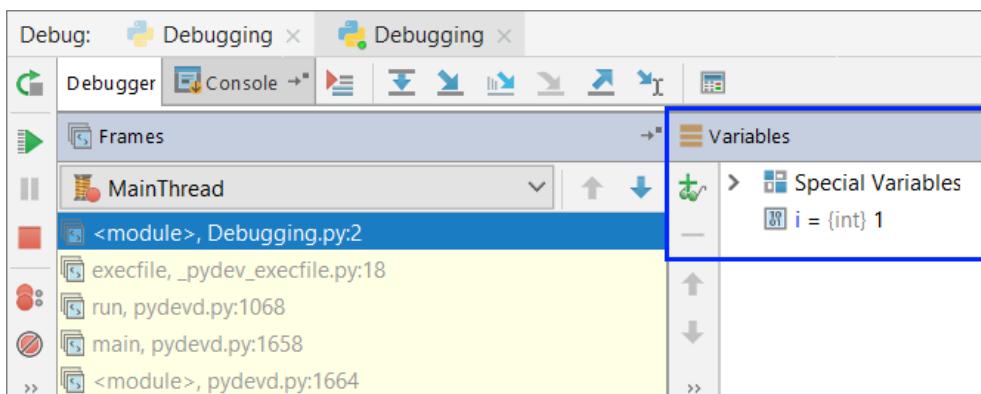
След стартиране на програмата виждаме, че тя **спира изпълнението си** на ред 2, където сложихме стопера (breakpoint). Кодът на текущия ред се **оцветява с жълт цвят** и можем да го **изпълняваме постъпково**. За да преминем на **следващ ред** използваме клавиш [F8]. Забелязваме, че кодът на текущия ред все още не е изпълнен. Изпълнява се, когато преминем на **следващия ред**:

```

1 for i in range(1, 100): i: 1
2 ● print(i)

```

От прозореца **Debugger** можем да наблюдаваме **промените по локалните променливи**. Той се отваря, когато започнем да дебъгваме. За да отворите прозореца ръчно, изберете **[View] -> [Tool Windows] -> [Debug]**:



## Справочник с хитрости

В тази секция ще припомним накратко **хитрости и техники** от програмирането с езика **Python**, разглеждани вече в тази книга, които ще са ви много полезни, ако ходите на изпит по програмиране за начинаещи:

### Вкаране на променливи в стринг (string)

```

text = "some text"
print(f"{text}")
# Това ще отпечата на конзолата "some text"

```

В случая използваме **placeholder – {x}**, където **x** е **името на променливата**, която искаме да покажем. Възможно е да използваме повече от една променлива, примерно:

```

text = "some text"
number = 5
print(f"{text} {number} {text}")
# Това ще отпечата "some text 5 some text"

```

В този пример забелязваме, че можем да подаваме **не само текстови променливи**. Също така можем да използваме дадена променлива **няколко пъти**.

## Закръгляне на числа

При нужда от закръгляне можем да използваме един от следните методи:

- **round(...)** – приема два параметъра – първият е **числото**, което искаме да закръглим, а вторият – цяло число, което задава **с колко символа след десетичния знак да се извърши закръглянето**. Закръглянето се извършва по основното математическо правило – ако десетичната част е по-малка от 5, числото се закръгля надолу и обратно: ако е по-голяма от 5 – нагоре:

```
first_number = 5.431
print(round(first_number, 2))
# Това ще отпечата на конзолата "5.43"

second_number = 5.539
print(round(second_number, 2))
# Това ще отпечата на конзолата "5.54"
```

- **math.floor(...)** – в случай, че искаме закръглянето да е винаги **надолу**. Важно е да отбележим, че тази функция закръгля до цяло число. Например, ако имаме числото 5.99 и използваме **math.floor(5.99)**, ще получим числото 5:

```
number_to_floor = 5.99
print(math.floor(number_to_floor))
# Това ще отпечата на конзолата "5"
```

- **math.ceil(...)** – в случай, че искаме закръглянето да е винаги **нагоре**. Тази функция също закръгля до цяло число. Например, ако имаме числото 5.11 и използваме **math.ceil(5.11)**, ще получим числото 6:

```
number_to_ceil = 5.11
print(math.ceil(number_to_ceil))
# Това ще отпечата на конзолата "6"
```

- **math.trunc(...)** – в случай, че искаме да **премахнем дробната част**. Например, ако имаме числото 2.63 и използваме **math.trunc(2.63)**, ще получим 2:

```
number_to_truncate = 2.63
print(math.trunc(number_to_truncate))
# Това ще отпечата на конзолата "2"
```

## Закръгляне чрез placeholder

```
number = 5.432424
print(f"{number:.2f}")
```

В случая след числото добавяме `:.2f`, което ще ограничи числото до 2 цифри след десетичния знак. Поведението ще е като на функцията `round(...)`. Трябва да имаме предвид, че числото преди буквата `f` означава до колко цифри след десетичния знак да е закръглено числото (т.е. може да е примерно `3f` или `5f`). Не забравяйте и **точката преди числото** – тя е задължителна.

## Как се пише условна конструкция?

Условната `if` конструкция се състои от следните елементи:

- Ключова дума `if`.
- Булев израз (условие).
- Тяло на условната конструкция.
- Незадължително: `else` клауза.

```
if условие:
    тяло
else:
    тяло
```

За улеснение може да използваме live template за `if` конструкция:

- `if` + [Tab]

## Как се пише for цикъл?

За `for` цикъл ни трябват няколко неща:

- Инициализационен блок, в който се декларира променливата брояч (`i`).
- Граници за повторение: `range(5)` са числата от `0` до `4` включително.
- Тяло на цикъла.

```
for i in range(5):
    тяло
```

За улеснение може да използваме live template за `for` цикъл:

- `for` + [Tab]
- или `iter` + [Tab]

## Какво научихме от тази глава?

В настоящата глава се запознахме как правилно да форматираме и именуваме елементите на нашия код, някои бързи клавиши (shortcuts) за работа в PyCharm, шаблони с код (Live Templates) и разгледахме как се дебъгва код.

# Заключение

Ако сте прочели **цялата** книга и сте решили всички задачи от упражненията и сте стигнали до настоящото заключение, заслужавате **поздравления!** Вече сте направили **първата стъпка** от изучаването на **професията на програмиста**, но имате още доста **дълъг път** докато станете **истински добри** и превърнете **писането на софтуер** в своя професия.

Спомнете си за **четирите основни групи умения**, които всеки програмист трябва да притежава, за да работи своята професия:

- Умение #1 – **писане на програмен код** (20% от уменията на програмиста) – покриват се до голяма степен от тази книга, но трябва да изучите още базови структури от данни, класове, обекти, функции, стрингове и други елементи от писането на код.
- Умение #2 – **алгоритмично мислене** (30% от уменията на програмиста) – покриват се частично от тази книга и се развиват най-вече с решаване на голямо количество разнообразни алгоритмични задачи.
- Умение #3 – **фундаментални знания за професията** (25% от уменията на програмиста) - усвояват се за няколко години с комбинация от учене и практикуване (четене на книги, гледане на видео уроци, посещаване на курсове и най-вече писане на разнообразни проекти от различни технологични области).
- Умение #4 - **езици за програмиране и софтуерни технологии** (25% от уменията на програмиста) - усвояват се продължително време, с много практика, здраво четене и писане на проекти. Тези знания и умения бързо отаряват и трябва непрестанно да се актуализират. Добрите програмисти учат всеки ден нови технологии.

## Тази книга е само първа стъпка!

Настоящата книга по основи на програмирането е само **първа стъпка** от изграждането на уменията на един програмист. Ако сте успели да решите **всички задачи**, това означава, че сте **получили ценни знания** за принципите на програмиране с езика **Python** на **базисно ниво**. Тепърва ви предстои да изучавате **по-задълбочено** програмирането, както и да развивате **алгоритмичното си мислене**, след което да добавите и **технологични знания** за езика Python и Django екосистемата, front-end уеб технологии (HTML, CSS, Angular, React, AJAX, HTML5) и още редица концепции, технологии и инструменти за разработка на софтуер.

Ако **не сте успели** да решите всички задачи или голяма част от тях, върнете се и ги решете! Помните, че за да **станете програмисти** се изискват много труд и **усилия**. Тази професия не е за мързеливци. Без **да практикувате сериозно** програмирането години наред, няма как да го научите!

Както вече обяснихме, първото и най-базово умение на програмиста е **да се научи да пише код** с лекота и удоволствие. Именно това е мисията на тази книга: да ви научи да кодите. Препоръчваме ви освен книгата, да запишете и [практическия курс "Основи на програмирането" в СофтУни](#), който се предлага напълно безплатно в присъствена или онлайн форма на обучение.

## Накъде да продължим след тази книга?

С тази книга сте поставили стабилни основи, благодарение на които ще ви е лесно да продължите да се развивате като програмисти. Ако се чудите как да продължите развитието си, помислете за следните няколко възможности:

- Да учите за **софтуерен инженер** в СофтУни и да направите програмирането своя професия.
- Да продължите развитието си като програмист **по свой собствен път**, например чрез самообучение или с някакви онлайн уроци.
- Да си **останете на ниво кодер**, без да се занимавате с програмиране по-сериозно.

## Професия "софтуерен инженер" в СофтУни

Първата, и съответно препоръчителната, възможност да овладеете цялостно и на ниво професията "софтуерен инженер", е да започнете своето обучение по **цялостната програма на СофтУни за подготовка на софтуерни инженери**: <https://softuni.bg/curriculum>. Учебният план на СофтУни е внимателно разработен от **д-р Светлин Наков и неговия екип**, за да ви поднесе последователно и с градираща сложност всички умения, които един софтуерен инженер трябва да притежава, за **да стартира кариера като разработчик на софтуер** в ИТ фирма.

## Продължителност на обучението в СофтУни

Обучението в СофтУни е с продължителност **2-3 години** (в зависимост от професията и из branите специализации) и за това време е нормално да достигнете добро начално ниво (junior developer), но **само ако учите сериозно** и здраво пишете код всеки ден. При добър успех един типичен студент **започва работа на средата на обучението си** (след около 1.5 години). Благодарение на добре развита партньорска мрежа **кариерният център на СофтУни** предлага **работка** в софтуерна или ИТ фирма на всички студенти в СофтУни, които имат много добър или отличен успех. **Започването на работа** по специалността при силен успех в СофтУни, съчетан с желание за работа и разумни очаквания спрямо работодателя, е почти гарантирано.

## Програмист се става за най-малко година здраво писане на код

Предупреждаваме ви, че **програмист се става с много усилия**, с писане на десетки хиляди редове код и с решаване на стотици, дори хиляди практически задачи, и отнема години! Ако някой ви предлага "по-лека програма" и ви обещава да

станете програмисти и да започнете работа за 3-4 месеца, значи или ви **льже**, или ще ви даде толкова ниско ниво, че **няма да ви вземат даже за стажант**, дори и да си плащате на фирмата, която си губи времето с вас. Има и изключения, разбира се, например ако не започвате от нулата или ако имате екстремно развито инженерно мислене или ако кандидатствате за много ниска позиция (например техническа поддръжка), но като цяло **програмист за по-малко от 1 година здраво учене и писане на код не се става!**

## Приемен изпит в СофтУни

За **да се запишете в СофтУни** е нужно да се явите на **приемен изпит** по "Основи на програмирането", върху материала от тази книга. Ако решавате с лекота задачите от упражненията в книгата, значи сте готови за изпита. Обърнете внимание и на няколкото глави за **подготовка за практически изпит по програмиране**. Те ще ви дадат добра представа за трудността на изпита и за типовете задачи, които трябва да се научите да решавате.

Ако задачите от книгата и подготвителните примерни изпити ви затрудняват, значи имате **нужда от още подготовка**. Запишете се на [бесплатния курс по "Основи на програмирането"](#) или преминете внимателно през книгата още веднъж отначало, без да пропускате да решавате **задачите от всяка една учебна тема!** Трябва да се научите **да ги решавате с лекота**, без да си помагате с насоките и примерните решения.

## Учебен план за софтуерни инженери

След изпита ви очаква **сериозен учебен план** по програмата на СофтУни за обучение на софтуерни инженери. Той е поредица от **модули с по няколко курса** по програмиране и софтуерни технологии, изцяло насочени към усвояване на фундаментални познания от разработката на софтуер и придобиване на **практически умения за работа** като програмист с най-съвременните софтуерни технологии. На студентите се предоставя избор измежду **няколко професии** и специализации с фокус върху C#, Java, JavaScript, PHP и други езици и технологии. Всяка професия се изучава в няколко модула с продължителност от 4 месеца и всеки модул съдържа 2 или 3 курса. Учебните занятия са разделени на **теоретична подготовка (30%)** и **практически упражнения, проекти и занимания (70%)**, а всеки курс завършва с практически изпит или практически курсов проект.

## Колко часа на ден отнема обучението?

Обучението за софтуерен инженер в СофтУни е **много сериозно занимание** и трябва да му отделите като **минимум поне по 4-5 часа** **всеки ден**, а за препоръчване е да посветите цялото си време на него. Съчетанието на **работка с учене** невинаги е успешно, но ако работите нещо леко с много свободно време, е добър вариант. СофтУни е подходяща възможност за **ученици, студенти и работещи други професии**, но най-добре е да отделите цялото си време за вашето образование и овладяването на професията. Не става с 2 или 4 часа на седмица!

Формите на обучение в СофтУни са **присъствена** (по-добър избор) и **онлайн** (ако нямате друга възможност). И в двете форми, за да успеете да научите предвиденото в учебния план (което се изисква от софтуерните фирми за започване на работа), е необходимо **здраво учене**. Просто **трябва да намерите време!** Причина #1 за букасуване по пътя към професията в СофтУни е неотделянето на достатъчно време за обучението: като минимум поне 20-30 часа на седмица.

## Софтуни за работещи и учащи другаде

На всички, които изкарат **отличен резултат на приемния изпит в СофтУни** и се запалят истински по програмирането и мечтаят да го направят своя професия, препоръчваме да се освободят от останалите си ангажименти и **да отделят цялото си време**, за да научат професията "софтуерен инженер" и да започнат да си изкарват хляба с нея.

- За **работещите** това означава да си напуснат работата (и да вземат заем или да си свият финансовите разходи, за да изкарат с по-нисък доход 1-2 години до започване на работа по новата професия).
- За **учащите** в традиционен университет това означава да си изместят силно фокуса към програмирането и практическите курсове в СофтУни, като намалят до минимум времето, което отделят за традиционния университет.
- За **безработните** това е отличен шанс да вложат цялото си време, сили и енергия, за да придобият една перспективна, добре платена и много търсена професия, която ще им осигури високо качество на живот и дългосрочен просперитет.
- За **учениците** от средните училища и гимназии това означава **да си сложат приоритет** какво е по-важно за тяхното развитие: да учат практическо програмиране в СофтУни, което ще им даде професия и работа или да отделят цялото си внимание на традиционната образователна система или да съчетават умело и двете начинания. За съжаление, често пъти приоритетите се задават от родителите и за тези случаи нямаме решение.

На всички, които **не могат да изкарат отличен резултат на приемния изпит в СофтУни** препоръчваме да набледнат върху по-доброто изучаване, разбиране и най-вече практикуване на учебния материал от настоящата книга. Ако не се справяте с лекота със задачите от тази книга, няма да се справяте и за напред при изучаването на програмирането и разработката на софтуер.

**Не пропускайте основите на програмирането!** В никакъв случай не трябва да взимате смели решения да напускате работата си или традиционния университет и да кроите велики планове за бъдещата си професия на софтуерен инженер, ако нямате отличен резултат на входния изпит в СофтУни! Той е мерило доколко ви се отдава програмирането, доколко ви харесва и доколко наистина сте мотивирани да го учите сериозно и да го работите след това години наред всеки ден с желание и наслада.

## Професия "софтуерен инженер" по ваш собствен път

Другата възможност за развитие след тази книга е да продължите да изучавате програмирането извън СофтУни. Можете да запишете или да следите **видео курсове**, които навлизат в по-голяма дълбочина в програмирането с **Python** или други езици и платформи за разработка. Можете да четете книги за програмиране и софтуерни технологии, да следвате онлайн ръководства (*tutorials*) и други онлайн ресурси - има безкрайно много бесплатни материали в Интернет. Запомнете, обаче, че най-важното по пътя към професията на програмиста е да правите практически проекти!

Без писане на много, много код и здраво практикуване, не се става програмист. Отделете си достатъчно време. Програмист не се става за месец или два. В Интернет ще намерите голям набор от свободни ресурси като книги, учебници, видео уроци, онлайн и присъствени курсове за програмиране и разработка на софтуер. Обаче, ще трябва да инвестирате поне година-две, за да добиете начално ниво като за започване на работа.

След като понапреднете, намерете начин или да започнете **стаж в някоя фирма** (което ще е почти невъзможно без поне година здраво писане на код преди това) или да си измислите **ваш собствен практически проект**, по който да поработите няколко месеца, даже година, за да се учате чрез проба-грешка.



Запомнете, че има много начини да станете програмисти, но всички те имат общая пресечна точка: **здраво писане на код и практика години наред!**

## Онлайн общности за стартиращите в програмирането

Независимо по кой път сте поели, ако ще се занимавате сериозно с програмиране, е препоръчително да следите специализирани **онлайн форуми, дискусионни групи и общности**, от които можете да получите помощ от свои колеги и да следите новостите от софтуерната индустрия.

Ако ще учате програмиране сериозно, **обградете се с хора**, които се занимават с **програмиране** сериозно. Присъединете се към **общности** от софтуерни **разработчици**, ходете по софтуерни конференции, ходете на събития за програмисти, намерете си приятели, с които да си говорите за програмиране и да си обсъждате проблемите и бъговете, намерете среда, която да ви помага. В София и в големите градове има бесплатни събития за програмисти, по няколко на седмица. В по-малките градове имате Интернет и достъп до цялата онлайн общност.

Ето и някои препоръчителни **ресурси**, които ще са от полза за развитието ви като програмист:

- <https://softuni.bg> - официален **уеб сайт на СофтУни**. В него ще намерите бесплатни (и не само) курсове, семинари, видео уроци и обучения по програмиране, софтуерни технологии и дигитални компетенции.
- <https://softuni.bg/forum> - официален **форум на СофтУни**. Форумът за дискусии на СофтУни е изключително позитивен и изпълнен с желаещи да помогат колеги. Ако зададете смислен въпрос, свързан с програмирането и изучаваните в СофтУни технологии, почти сигурно ще получите смислен отговор до минути. Опитайте, нищо не губите.
- <https://www.facebook.com/SoftwareUniversity> - официална **Facebook страница на СофтУни**. От нея ще научавате за нови курсове, семинари и събития, свързани с програмирането и разработката на софтуер.
- <http://www.introprogramming.info> - официален уеб сайт на **книгите "Въведение в програмирането"** със **C#** и **Java** от д-р Светлин Наков и колектив. Книгите разглеждат в дълбочина основите на програмирането, базовите структури от данни и алгоритми, ООП и други базови умения и са отлично продължение за четене след настоящата книга. Обаче **освен четене, трябва и здраво писане**, не забравяйте това!
- <http://stackoverflow.com> - **Stack Overflow** е един от **най-големите** в световен мащаб дискусионни форуми за програмисти, в който ще получите помощ за всеки възможен въпрос от света на програмирането. Ако владеете английски език, търсете в StackOverflow и задавайте въпросите си там.
- <https://fb.com/groups/bg.developers> - групата "**Програмиране България @ Facebook**" е една от най-големите онлайн общности за програмисти и дискусии по темите на софтуерната разработка на български език във Facebook.
- <https://www.meetup.com/find/tech> - потърсете **технологични срещи (tech meetups)** около вашия град и се включете в общностите, които харесвате. Повечето технологични срещи са безплатни и новобранци са добре дошли.
- Ако се интересувате от ИТ събития, технологични конференции, обучения и стажове, разгледайте и по-големите **сайтове за ИТ събития** като <http://iteventz.bg> и <http://dev.bg>.

## Успех на всички!

От името на целия авторски колектив ви **пожелаваме неспирни успехи в професията и в живота!** Ще сме невероятно щастливи, ако с наша помощ сте се **запалили по програмирането** и сме ви вдъхновили да поемете смело към професията "**софтуерен инженер**", която ще ви донесе добра работа, която да работите с удоволствие, качествен живот и просперитет, като и страховити перспективи за развитие и възможности да правите значими проекти с вдъхновение и страсть.



В ръцете си държите нещо повече от **книга за програмиране**, учебник или учебно помагало. Това съвременно образователно пособие ви повежда по **първите стъпки към програмирането** чрез малко текст и **много код**, наситено с примери и внимателно подбрани **практически задачи** със система за моментално **автоматично оценяване**.

Учебното съдържание е разработено лично от **д-р Светлин Наков**, който през 15-годишния си опит с обучението на софтуерни инженери помага на **над 70 000 души** да навлязат в програмирането и намира как да поднася информацията на **малки порции**, с много практика и с **нарастваща сложност**.

Запомнете, че програмиране се учи с **много писане на код и усилено решаване на задачи** и не става само с четене на книги, така че преминете старателно през **упражненията**. Успех!

Сайт: [python-book.softuni.bg](http://python-book.softuni.bg)



## Авторски колектив:

**Бончо Вълков**  
**Венцислав Петров**  
**Владимир Дамяновски**  
**Илия Илиев**  
**Йордан Даракчиев**  
**Мартин Царев**  
**Миглен Евлогиев**  
**Милена Ангелова**  
**Мирела Дамянова**  
**Николай Костов**  
**Петър Иванов**  
**Петя Господинова**  
**Светлин Наков**  
**Таня Евтимова**  
**Таня Станева**  
**Теодор Куртев**  
**Христо Минков**

ISBN 978-619-00-0806-4



9 786190 008064 >