

## Ще одна блок-схема

Замість повторювати приклад із паспортом, вирішив пошукати вам ще щось із серії "як зробити". Google запропонував ["як зробити зміст у ворді"](#) -- підозрюю, це і справді буде комусь корисно.

Для того, щоб не заплутатися в численних подробицях, є сенс складати [блок-схему](#) алгоритму в декілька ітерацій: почати з найбільш загального і поступово його уточнювати.

Фу! Блок-схеми! Та вони ще під перфокарти заточені! Не для того ми на курс записувалися, хочемо кодити! -- з одного боку так, а з іншого -- більш зручного способу візуального представлення алгоритмів я не знаю. [Діаграми діяльності UML](#) не пропонувати -- це майже ті самі блок-схеми.

Спочатку пройдемо по заголовках найвищого рівня -- це буде наша чернетка. Їх всього 2, але не забуваємо, що у кожної програми і у кожного алгоритму є якась точка входу (початок роботи) та точка виходу:



Далі пройдемо по заголовках наступного рівня і замінимо початкові дії новими, з яких вони насправді складаються.

При цьому з контексту видно, що на етапі позначення елементів можна позначити заголовки, а потім до них додати інші елементи. Наприклад, (згадуються ГОСТи оформлення документації) підписи до таблиць та зображень. Тобто ці дії можна виконувати послідовно.

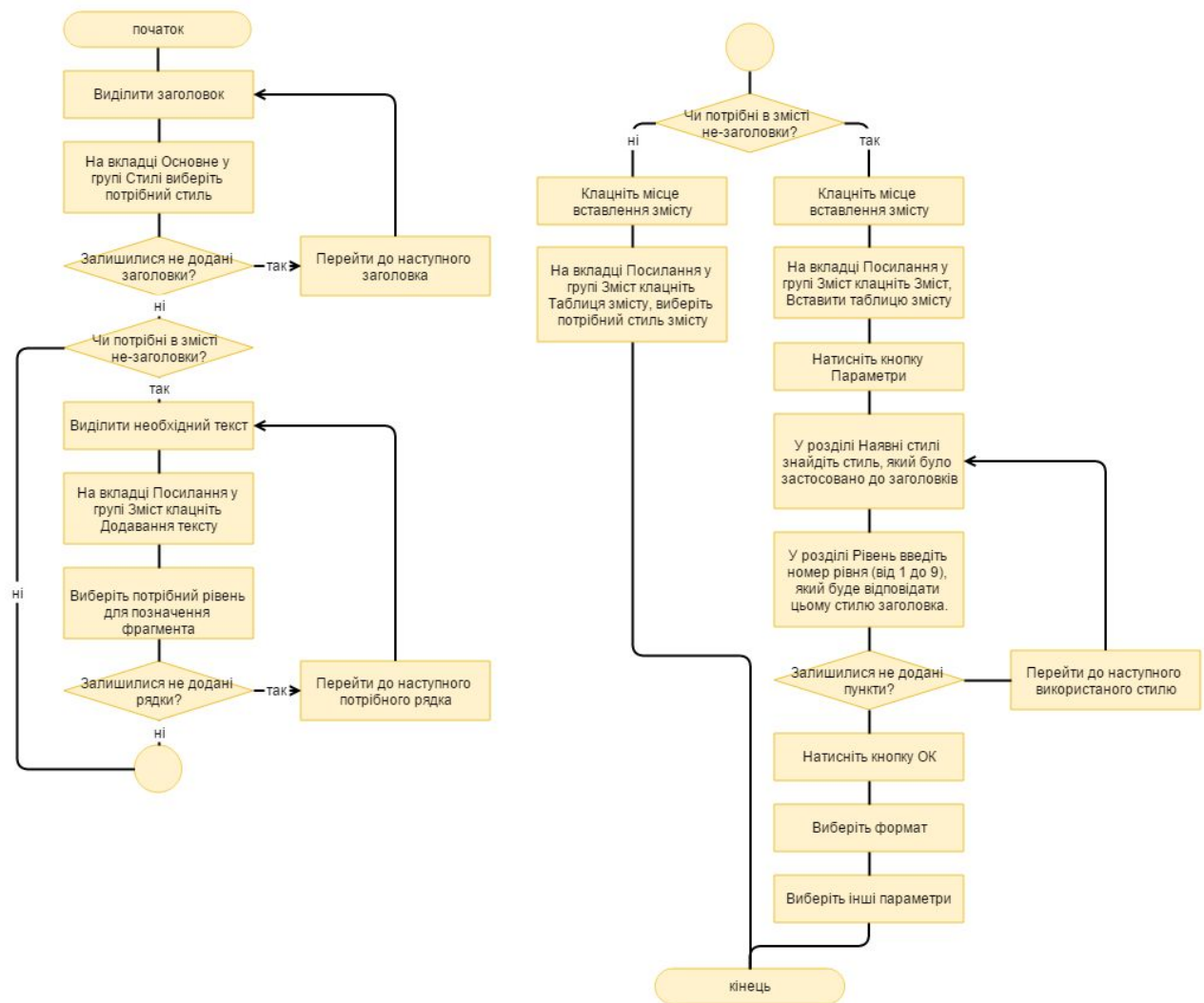
А от при створенні самого змісту 2 запропоновані дії конфліктують, отже треба виконати одну або іншу -- це явне розгалуження. Звичайно, 2-й варіант функціонально перекриває 1-й і можна було б залишити лише його. Але, судячи з

опису, 1-й значно простіший і у випадку, якщо в змісті присутні лише заголовки, легше створити меню в "2 кліки" ніж мучитися з додаванням стилів вручну.

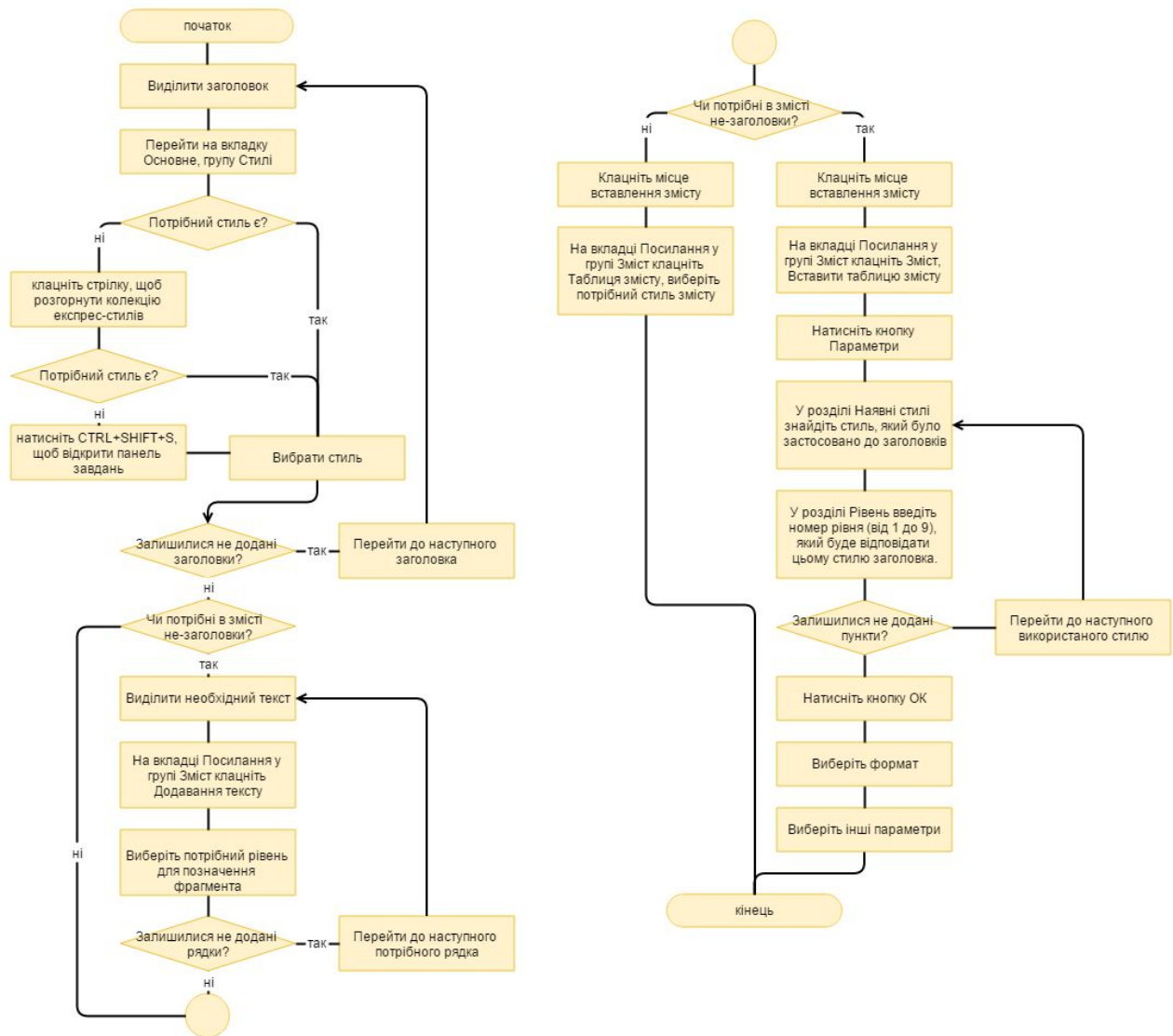


Тепер всі 4 дії можна деталізувати на основі списків дій у статті. Для опрацювання послідовності елементів (виділення всіх заголовків, всіх потрібних рядків, додавання всіх стилів) використовуються цикли.

Крім того, схема стає надто довгою і для більш оптимального її розміщення на екрані або на папері час розбити її на фрагменти. Для позначення переходу між ними використовується круглий з'єднувальний елемент. Якщо таких розривів декілька, їх необхідно пронумерувати і відповідно позначити з'єднання одного розриву одним номером.



Крім того, у нас залишилися неопрацьовані примітки. В якості прикладу, розпишемо першу з них. При цьому дію "перейти на вкладку Основне ... вибрати стиль" доведеться розбити на 2, так як "вибрати стиль" в усіх випадках -- одна й та сама дія.



Крім приміток, можна розбити на елементарні дії всякі переходи по меню та формах і, звичайно, пункт "Виберіть інші параметри". У випадку виконавця-комп'ютера максимальним рівнем деталізації будуть оператори мови програмування. (В принципі можна розбити далі на машинні команди, які їх реалізують, але все-таки ми не збираємося програмувати машинними кодами).

У псевдокодi це виглядатиме приблизно наступним чином... Вже чую гнівні крики: Знову динозаври! Це ж курс 2015 року! Ще моя бабуся мені розповідала про псевдокод! Подякуйте бабусі -- на мою думку, він справді корисний. Звичайно не для роботи, але в якості ілюстрації для навчання. Це вже майже програма, лише не прив'язана до синтаксису конкретної мови програмування. Така собі "абстрактна мова у вакуумі":

## 1. Виділити заголовок

2. Перейти на вкладку Основне, групу Стили
3. Якщо Потрібного стилю нема:
  - 3.1. Клацніть стрілку, щоб розгорнути колекцію експрес-стилів
  - 3.2. Якщо Потрібного стилю нема:
    - 3.2.1. Натисніть CTRL+SHIFT+S, щоб відкрити панель завдань
4. Виберіть стиль
5. Якщо залишилися неопрацьовані заголовки, повернутися до пункту 1
- ...

ОК, попався, намагаючись зробити синтаксис свого "псевдокоду" більш схожим на Python. Але із різними циклами не вийшло. В класичному варіанті не використовується вкладеність, а вказуються кроки, куди треба перейти:

1. Виділити заголовок
2. Перейти на вкладку Основне, групу Стили
3. Якщо Потрібний стиль є, перейти на крок 7, інакше перейти на крок 4
4. Клацніть стрілку, щоб розгорнути колекцію експрес-стилів
5. Якщо Потрібний стиль є, перейти на крок 7, інакше перейти на крок 6
6. Натисніть CTRL+SHIFT+S, щоб відкрити панель завдань
7. Виберіть стиль
8. Якщо залишилися неопрацьовані заголовки, перейти на крок 1, інакше перейти на крок 9
- ...

Погодьтеся, 1-й варіант виглядає краще. Непогана ілюстрація того, чим Python відрізняється від старих (маю на увазі зовсім старих) мов програмування.

.....

# Інтерпретатори проти компіляторів

## МАШИННИЙ КОД

Як уже зазначалося, для безпосереднього виконання комп'ютером програма має бути представлена у формі [машинного коду](#). Такий запис містить лише номери команд процесора, необхідні дані та адреси комірок пам'яті. І виглядає приблизно наступним чином (для зручності двійкові дані найчастіше записуються у шістнадцятковій формі, де 2 символи відповідають 1 байту даних -- саме тому шістнадцяткова система числення є такою популярною в програмуванні):

```
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C 6F 2C 20 57 6F  
72 6C 64 21
```

## АСЕМБЛЕРИ

Для написання таких програм застосовуються [мови асемблера](#), які дозволяють записувати команди замість числової форми у текстовій (MOV, ADD, IN, OUT) та містять деякі найпростіші засоби для полегшення написання програми. Тим не менше, навіть в такому "прикрашеному" вигляді програма залишається надзвичайно близькою до машинного коду і тому мови асемблера відносять до низькорівневих мов програмування (тобто таких, що близькі до рівня машинного коду).

Незважаючи на незручність написання, такий код виконується з найвищою швидкістю і може бути максимально оптимізований, так як програміст має доступ буквально до кожного біту у ньому. Тому найчастіше низькорівневі мови застосовуються для програмування мікросхем та окремих дій у прикладних програмах, де швидкодія є критичною.

Асемблер же -- це програма, яка перетворює код, написаний мовою асемблера (вибачте за тавтологію), остаточно у машинний код. Але часто і саму мову називають скорочено "асемблером".

## ТРАНСЛЯТОРИ

Більшість же програм пишеться на високорівневих алгоритмічних мовах програмування. Як я і говорив, такі мови зазвичай мають складний синтаксис, використовують слова-оператори близькі до людської мови і -- що найголовніше -- реалізують алгоритмічні структури для простого і зрозумілого запису програми.

З іншого боку, для виконання комп'ютером така програма має бути перетворена на машинний код або хоча б переписана низькорівневою мовою (а далі вже асемблер забезпечить її розуміння машиною). Причому команди високорівневої мови програмування можуть бути досить складними і відповідати кільком або навіть

кільком десяткам машинних команд. Цей процес перетворення називається трансляцією, а програми, які його виконують -- трансляторами.

В цьому контексті транслятори за принципом роботи поділяються на 2 типи: [компілятори та інтерпретатори](#).

## КОМПІЛЯТОРИ

Компілятор зчитує одразу всю програму і переписує її машинним кодом або мовою асемблеру. Процес трансляції, який при такому підході називається компіляцією, відбувається один раз і результат зберігається. Якщо код програми пізніше буде змінено, її необхідно буде перекомпілювати.

Скомпільована програма прив'язується до операційної системи і набору команд процесора, тому не завжди може бути перенесена і виконана на іншому комп'ютері. З іншого боку вона є "готовою до вживання" і може бути швидко виконана на тій самій або аналогічній машині: з точки зору користувача -- просто клацнути і запустити, з точки зору комп'ютера -- просто прочитати набір команд і виконати.

## ІНТЕРПРЕТАТОРИ

Інтерпретатор зчитує вихідний код програми по одній інструкції і, в найпростішому випадку, одразу намагається їх "перекладати" та виконувати. Це дозволяє програмісту швидше перевіряти виконання програми та знаходити помилки в коді. Крім того, така програма може бути легко перенесена на іншу машину і, якщо там є потрібний інтерпретатор, виконана ним -- незалежно від операційної системи та процесора. А різницю між особливостями різних комп'ютерів покриває сам інтерпретатор, який, звичайно, буде трохи відрізнятися.

Логічно, що в такій схемі виконання програми буде займати трохи більше часу -- так як при цьому кожного разу відбувається аналіз коду та його перетворення.

Тому для підвищення швидкодії більшість сучасних інтерпретаторів насправді працює за змішаною схемою, спочатку транслюючи вихідний код програми у деяку проміжну форму -- так званий [байт-код](#). Він є кодом нижчого рівня і ближчий до асемблеру, але машинно-незалежний -- тому виконується не безпосередньо комп'ютером, а деякою віртуальною машиною, яка входить до складу інтерпретатора. Це дозволяє, за відсутності змін в оригінальній програмі, не перечитувати її повністю, а використовувати байт-код як "напівфабрикат" для роботи. Виконання байт-коду все одно повільніше ніж машинного коду, але такий підхід є компромісом, що намагається поєднати переваги інтерпретації та компіляції.

Як ви пам'ятаєте, Python -- інтерпретована мова програмування. І також створює байт-код для більш швидкої роботи. Наступного разу зверніть увагу на файли з

розширенням .рус, які з'являються у папці із текстами ваших програм під час їх виконання, -- це він і є.

## Арифметика у Python

### ПРОСТІ ТИПИ ДАНИХ

По суті все, чим займається комп'ютер як обчислювальна машина, -- це обробка [даних](#). У високорівневих мовах програмування всі дані належать до якихось вбудованих типів. Найпростішими з них є рядки та числа (цілі та дійсні розглядаються окремо, так як реалізуються в комп'ютері по-різному), які просто являють собою текстові фрагменти або числові значення.

Всі вони відображаються в python по-різному і, якщо ви задасте змінній якийсь значення за його зовнішнім виглядом інтерпретатор може визначити тип даних, який мається на увазі. У випадку, коли інтерпретатор не в змозі самостійно правильно визначити тип, програміст може конвертувати значення у необхідний тип даних примусово. Для цього застосовуються вбудовані функції, які, подібно до математичних функцій, приймають аргумент -- дані, які потрібно конвертувати, та повертають перетворене значення необхідного типу.

Int -- цілі числа: 1, 2, 0, -10, 9999 і т.д. Відображаються просто як числа. Для перетворення будь-якого значення на ціле число використовується функція `int()`:

```
x_float = 1.0
x_int = int(x_float)
```

Float -- дійсні числа: 1.0, 2.543, 0.0, -0.99999, 123123123.4213 і т.д. Виглядають як 2 числа (ціла і дробова частина), розділені крапкою. Для перетворення будь-якого значення на ціле число використовується функція `float()`:

```
x_str = '1'
x_float = float(x_str)
```

Str -- рядки: 'High hopes', "The great gig in the sky", "When You're In" і т.д. Виглядають як текстові фрагменти будь-якої довжини -- від пустого рядка і до нескінченності (при цьому, зверніть увагу, що консоль/термінал, у якій ми вводимо рядок має мати власне обмеження довжини команди і просто не дозволить ввести надто великий рядок за 1 раз) -- взяті в одинарні чи подвійні лапки. Для перетворення будь-якого значення у рядок використовується функція `str()`:

```
x_int = 1
x_str = str(x_int)
```



Звичайно, не всі дані можна привести до будь-яких типів. Пропоную вам поекспериментувати із цим самостійно.

## АРИФМЕТИЧНІ ДІЇ

### Числа

Для чисел (як цілих так і дійсних) передбачено стандартні математичні операції: додавання (+), віднімання (-), множення (\*) та ділення (/). А крім того піднесення у степінь (\*\*), ділення націло (//) та взяття остачі (%) від ділення націло (всі ж пам'ятають, [що це таке?](#)). Першим у виразі обчислюється піднесення у степінь, потім операції множення, ділення та остачі від ділення, потім додавання та віднімання. Для зміни порядку проведення операцій, як і в математиці, використовуються круглі дужки.

```
print (2+2)**2 % 10
```

- виведе 2

При цьому, якщо хоч одне із значень у виразі є дійсним числом, для проведення розрахунків інтерпретатору доведеться конвертувати значення, які з ним взаємодіють, і результат також буде дійсним, навіть за відсутності у нього дробової частини.

```
print (2+2)**2 % 10.0
```

- виведе 2.0

І навпаки, якщо операнди є цілими, результат також буде цілим. Таким чином, ділення двох цілих чисел фактично також є операцією "ділення націло":

```
print 10/3
```

- виведе 3

І ще раз звертаю вашу увагу: перетворення типів при арифметичних обчисленнях виконується оператором лише у разі необхідності -- якщо один з двох [операндів](#) дії є дійсним, або ви примусово вказуєте, що необхідно конвертувати тип даних. Тому за цим слід уважно слідкувати:

```
print 3/2 * 1.0
```

- виведе 1.0

```
print 3.0/2 * 1
```

- виведе 1.5

### Рядки

Рядки можна додавати між собою -- ця операція називається конкатенацією.

```
print 'q' + 'w'  
- виведе 'qw'
```

А ще (несподівано) множити на ціле число X. Результатом цієї операції є початковий рядок, повторений X разів.

```
print 'q' * 3  
- виведе 'qqq'
```

Інші арифметичні операції для рядків невизначені.

## МАТЕМАТИЧНІ ФУНКЦІЇ

Крім арифметичних операцій будь-яка мова програмування містить також деякий набір математичних функцій. Аналогічно до функцій перетворення типів даних, вони приймають числові аргументи та повертають розраховане значення -- результат.

### Вбудовані функції

Деякі з них вбудовані в мову і доступні в будь-якому місці програми. Це

- `abs(x)` -- модуль від числа
- `bin(x)` -- переведення числа у двійкову систему числення
- `hex(x)` -- переведення числа у шістнадцяткову систему числення
- `max(x,y)` -- пошук максимуму з 2 чисел, також може приймати будь-яку кількість аргументів
- `min(x,y)` -- пошук мінімуму з 2 чисел, також може приймати будь-яку кількість аргументів
- `round(x)` -- округлення числа
- `round(x,y)` -- округлення числа x із вказаною точністю -- у знаків після коми

### Модуль `math`

Більшість математичних функцій недоступні в програмі з самого початку і для використання вимагають підключення стандартного модуля `math` -- фактично бібліотеки для інтерпретатора, в якій описано, яким чином вони мають обчислюватися (як саме це працює, ми розглянемо докладніше на наступних лекціях). Після підключення `math` на початку вашої програми ви можете вільно використовувати їх в обчисленнях, викликаючи як `math.<ім'я функції>(<аргументи функції>)`:

```
import math  
x = 16  
print math.sqrt(x)  
- виведе 4.0
```

Модуль `math` містить:

- 2 константи `math.pi` та `math.e`
- функції округлення чисел: `math.ceil(x)` (округлення "вверх"), `math.floor(x)` (округлення "вниз")
- степеневі та логарифмічні функції: `math.pow(x)`, `math.exp(x)`, `math.log(x)`, `math.log(x, y)`, `math.log10(x)`, `math.sqrt(x)`
- тригонометричні функції: `math.sin(x)`, `math.cos(x)`, `math.tan(x)`, `math.asin(x)`, `math.acos(x)` та інші (включаючи гіперболічні)
- функції для переведення градусів та радіан: `math.degrees(x)` та `math.radians(x)`
- та деякі інші

Більш повний список функцій модуля `math` можна знайти, наприклад, [тут](#) або в англomовній [документації](#).

## Командний рядок

Впевнений, ви не обмежитеся лише інтерактивним режимом роботи і будете зберігати свої програми як окремі файли. Все, що ми з вами напишемо протягом курсу -- це консольні застосування. Тобто програми, які працюють лише у командному рядку і не мають графічного інтерфейсу користувача.

Консольну програму можна запустити, як і будь-яку іншу, просто подвійним кліком на її файлі. При цьому операційна система відкриє вікно консолі, виконає в ній програму і закриє вікно. Якщо у програмі не передбачене "інтерактивне" введення даних користувачем, все відпрацює дуже швидко і ви навіть не встигнете роздивитися, що там відбулося. Тому зручніше запустити консоль самотушки і викликати програми в ній, щоб стежити за всім виведенням даних.

Нагадую, що для цього необхідно:

- для Windows: відкрити меню "пуск", "виконати", ввести `"cmd"` та натиснути Ентер або "ОК".
- для Linux та MacOS: знайти і запустити програму, яка називається "Термінал" або "Консоль" -- точна назва може відрізнятись, в залежності від виду та версії вашої операційної системи.

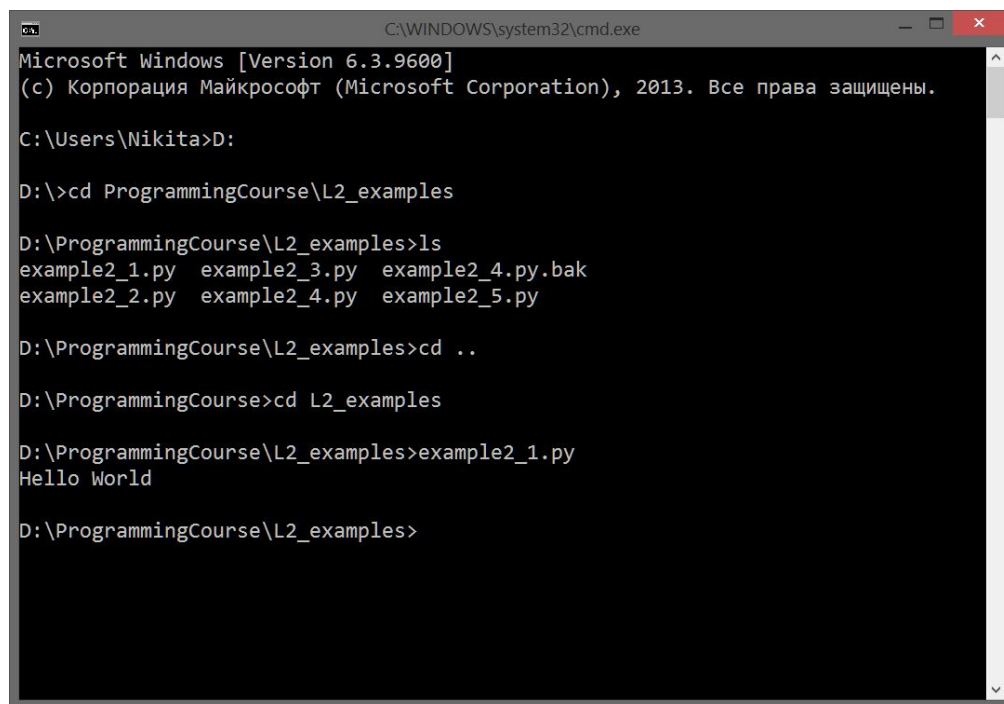
В запущеному "чорному" середовищі ви маєте доступ до всіх функцій операційної системи, лише без графічного інтерфейсу, а у форматі [командного рядка](#). Це значить, що ви можете вводити команди, а система буде їх виконувати.

Повний список команд можна отримати, виконавши команду `help`. А найчастіше вам будуть потрібні наступні:

`cd` -- зміна поточної папки. Ім'я папки для переходу може бути задане як абсолютно (з переліком всіх батьківських папок через `/`, для Windows `\`) так і відносно поточної папки. Для повернення "на рівень вгору" використовується позначення `..` (дві крапки). Користувачі Windows, зверніть увагу: таким чином можна подорожувати лише в межах одного диску, для переходу між дисками слід ввести літеру диску (C, D, E) з двокрапкою.

`ls` -- перегляд вмісту поточної папки. Без графічного інтерфейсу з незвички легко заблукати, тому використовуйте цю команду для того, щоб "піддивитися", куди рухатися далі.

В цьому місці було б добре вставити посилання на якісні статті про роботу з командним рядком. Але я поки не знайшов нічого краще ніж зовсім-зовсім прості: [для Windows](#) і [для Linux](#) (користувачі MacOS, не ображайтеся: всі поради для "лінуксоїдів" стосуються також і вас :-)). Тож, якщо у вас є корисні матеріали на цю тему -- діліться ними у обговоренні.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) Корпорация Майкрософт (Microsoft Corporation), 2013. Все права защищены.

C:\Users\Nikita>D:

D:\>cd ProgrammingCourse\L2_examples

D:\ProgrammingCourse\L2_examples>ls
example2_1.py  example2_3.py  example2_4.py.bak
example2_2.py  example2_4.py  example2_5.py

D:\ProgrammingCourse\L2_examples>cd ..

D:\ProgrammingCourse>cd L2_examples

D:\ProgrammingCourse\L2_examples>example2_1.py
Hello World

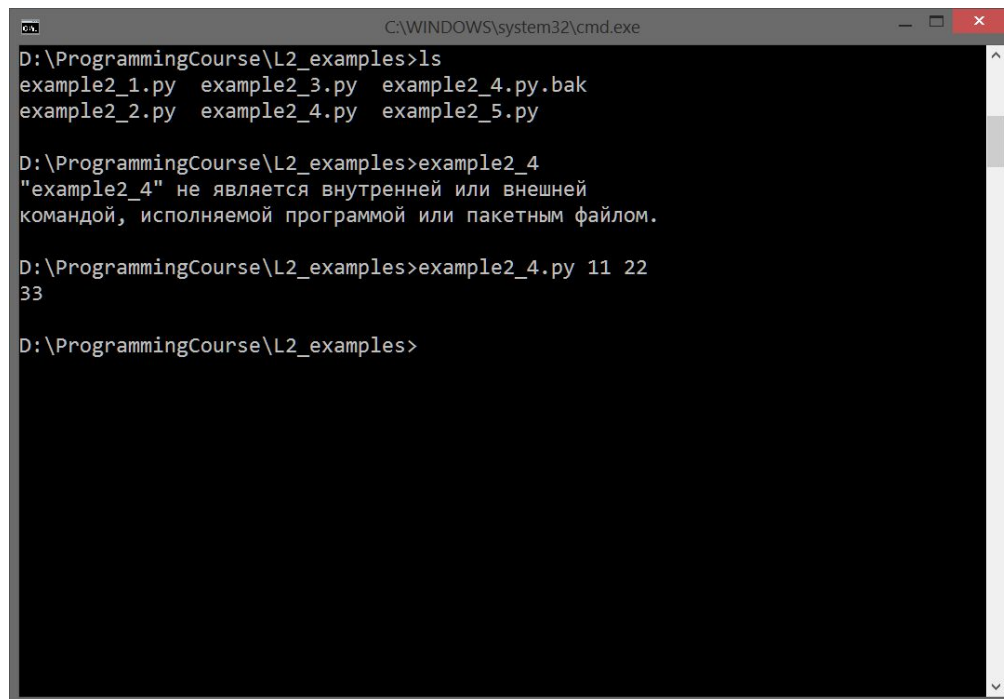
D:\ProgrammingCourse\L2_examples>
```

Для того, щоб викликати програму з консолі/терміналу необхідно звернутися до неї за іменем файлу -- логічно. Це вірно і для програм вже встановлених на комп'ютері, і для програм написаних вами, в тому числі на python. При цьому операційна система спробує знайти файл з такою ім'ям в поточній папці, а якщо там немає, то у папках, перерахованих у її змінній Path (для цього і слугувала опція "Add

python.exe to Path" при установці інтерпретатора). Або ж ви можете при виклику вказати повний шлях до програми -- і операційна система шукатиме її саме там.

А тепер -- те, заради чого насправді писалася дана стаття!

Запуск програм з консолі насправді є функціональнішим, ніж з графічного інтерфейсу. Так як при цьому ви можете передати програмі, яку запускаєте, додаткові значення-параметри. Сама ж програма має змогу зчитати ці параметри і відреагувати на них так, як це запрограмовано її авторами. Аналогічно, переходячи від папки до папки в консолі, ми передаємо команді `cd` параметр -- адресу папки, в яку хочемо потрапити. Це реалізовано трохи інакше, але для кінцевого користувача працює схожим чином.



```
C:\WINDOWS\system32\cmd.exe
D:\ProgrammingCourse\L2_examples>ls
example2_1.py  example2_3.py  example2_4.py.bak
example2_2.py  example2_4.py  example2_5.py

D:\ProgrammingCourse\L2_examples>example2_4
"example2_4" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.

D:\ProgrammingCourse\L2_examples>example2_4.py 11 22
33

D:\ProgrammingCourse\L2_examples>
```

Ці параметри називаються також аргументами (по аналогії з аргументами функції) або аргументами командного рядка. Вони розділяються між собою пробілами. А, якщо в якості параметра нам необхідно передати рядок з кількох слів, він береться у лапки.

І, звісно, ми можемо прочитати ці параметри з нашої програми. Для цього лише необхідно підключити модуль (ще одну бібліотеку функцій) `sys`. Він містить багато всього, але нас буде цікавити лише його змінна `sys.argv`, в якій зберігаються аргументи командного рядка, з якими запущено програму. Це -- змінна складного типу даних, яка може містити всередині інші різні дані, (ми ще повернемося до них пізніше) і кожен аргумент знаходиться в ній у окремій комірці.

А так, нагадую, виглядає наш приклад із відео-лекції, який зчитує аргументи командного рядка та використовує їх у обчисленнях:

```
import sys
x = int(sys.argv[1])
y = int(sys.argv[2])
print x + y
```