Вероятности, статистика и алгоритми

Както видяхме в предишните теми, понятия като *събитие* и *вероятност* се коренят в теорията на множествата и математическата логика, което ги свързва чисто идейно и с концепцията за алгоритъм. Тук обаче няма да говорим за тази абстрактна връзка, а вместо това ще се научим как да използваме възможностите на широко достъпните и бързи процесори за да провеждаме експерименти. Тази практическа необходимост се усеща още повече в статистиката, където реалната обработка на данни включва обикновено десетки или стотици хиляди еднотипни операции, за които би било разточително да се хаби човешки ресурс.

Типове данни и операции в Python 3

За да илюстрираме някои от идеите, разгледани в този курс, така че да не звучат абстракно и отвлечено, ще се възползваме от един от най-простите и интуитивни като синтаксис програмни езици, който същевременно предлага и много големи технически предимства, особено при работа с данни, а при това е и безплатен. Програмният интерпретатор заедно с удобна интерактивна среда и множество специализирани библиотеки може да бъде свален и инсталиран от следния линк:

https://www.anaconda.com/products/individual

като самият процес отнема малко време, но е доста лесен. По-старата версия Python 2 също е достъпна, ние обаче ще работим с последната. След като инсталирате, стартирайте Jupiter Notebook и изберете New → Python 3. В браузъра ще се отвори страница с поле за писане на код и няколко активни бутона над него, тяхната функция е ясна от наименованията им. Командите, които пишем в полето, се четат и изпълняват от интерпретатора, съответно ако даваме невалидни команди, получаваме съобщение за грешка, поради което е добре да се познава синтаксиса на езика, но при Python той е изключително прост и се учи в движение. По-интересно е какво бихме поискали от компютъра си: можем например да го караме да чете и пише вместо нас, да подбира, обработва и анализира информация достъпна в интернет и да взема решения, базирани на тези анализи, както и да провежда виртуални разговори с хора или други компютри.

Нека обаче започнем с нещо просто: присвояване на стойност на дадена променлива и комуникация с потребителя. Основните типове данни в Python, които ще използва са логически (Boolean: True, False), числови (int, float, complex), масиви (list, tuple, set) и текст (string). Последователност от действия с тях, която се изпълнява за крайно време и води до определен резултат, ще наричаме програма. Ако въведем например в полето за код x = 1, и натиснем бутона Run (или Shift+Enter на клавиатурата), сме изпълнили един алгоритъм, който присвоя-

ва на промеливата x целочислена стойност, равна на 1. Само че видимо за нас нищо не се случва. Това е така, защото не сме поискали нищо повече от това да присвоим съответната стойност. Ако добавим още един ред код, в който се казва **print**(x) и отново изпълним програмата, ще можем да видим присвоената стойност. За да разберем разликата между различните числови типове, нека въведем в първия ред на програмата x = **float**(1), тогава на екрана ще се изпише стойността 1.0, тъй като x вече се интерпретира като десетична дроб, съответно при **complex**(1), ще видим стойност $1 + 0j$, където j е стандартното означение за имагинерната единица. Съответно цялата част на число тип float се дава с командата **int**(), а закръглянето му до най-близкото цяло, с **round**(). Тип complex не може да бъде превърнат в int или float. Нашата промелнива може да присвои стойност текст, т.е. символен низ (string), като за целата се поставят кавички, например x = 'zdr ko pr', което ще се изпише на екрана с командата **print**(x). Както числовите типове, така и символните низове (string) могат да бъдат събирани: пробвайте например x = 2 + 3, или x = 'zdr ' + 'ko ' + 'pr'. С Python можем лесно да присвояваме и стойност, въведена от потребителя при самото изпълнение на програмата с командата **input**, например x = **input**('Здрасти, как се казваш?') отваря активно поле, и след това придава въведения в него текст като стойност на променливата x. За да се уверим в това, можем да продължим програмата като поискаме тази променлива да бъде включена в някакво стандартно изречение като **print**('Хубаво име, искам и аз да се казвам ' + x + '!').

Алгоритмите стават по-интересни когато в тях участват и логически, наричани още булеви (Boolean) промелниви, които вземат стойност True (истина) или False (лъжа). Всяко твърдение, което подлежи на проверка, може да бъде използвано като булева промелнива, така например 1 > 2 приема стойност False, тъй като е невярно, докато излъгнението на кода **print**(2>1) ще изпише на екрана True. Други релации между променливи, които дават логически твърдения са >= , <=, == и !=, като последните две означават съответно равенство и неравенство. Обърнете внимание на разликата между x = 2, което присвоява на променливата x стойност 2 и x == 2, което представлява твърдение, че x е равно на 2 и съответно може да бъде вярно или невярно. Обединението и сечението на подобни твърдения се дава с операторите **or** и **and**. Например A **and** B е вярно твърдение само ако A и B са едновременно верни, съответно A **or** B - когато поне едно от двете има стойност True. Понякога е удобно да използваме също и израза x **in** **range**(m,n), който е еквивалентен на x >= m **and** x < n. При комбиниране на повече твърдения скобите указват приоритет на действията, например A **and** B **and** C няма нужда от скоби, но A **and** (B **or** C) е различно от (A **and** B) **or** C, както видяхме в първия урок за събития и вероятности (разкриването на скобите е като при събиране и умножение). В алгоритмите булевите променливи се ползват в конструкции от вида **if** A: команда1 **else**: команда2, които определят различен отговор на програмата според това дали A е вярно твърдение или не.

```
In [ ]: age = input("Age: ")
        sex = input("Sex: ")
        location = input("Location: ")
        if int(age) in range(18,30) and sex == "female" and location == "Sofia":
            print("Hey, babe! You wanna send me a pic?")
        else:
            print("Not interested!")
```

Краткият код по-горе например описва поведението на един много разгонен чатбот, който подканва потребителя да въведе последователно възраст, пол и местоположение, след което на базата на тази информация взема решение какво съобщение да изпрати - да поиска снимка или да обяви, че не е заинтересован. Обърнете внимание на синтаксиса: командите след **if** A**:** се описват на нов ред и с отместване (tab) равно на четири интервала, същото важи за **else:** и други подобни оператори, с които предстои да се запознаем. Ако не спазите това правило, ще получите програма, която или не работи, или дава грешни резултати. Още една особеност е, че се наложи променливата age да бъде превърната от тип string, както се интерпретира всеки **input**, в тип int за да бъде проверено дали принадлежи на интервала $[18, 29]$ (горната граница при **range** не се включва).

Дотук основно се запознахме с част от възможностите за вход и изход на данни. Съществуват, разбира се, и много други: може да се окаже път за достъп до файл на твърдия диск или в интернет, откъдето Python да прочете и обработи нужната информация, след което да създаде свой файл и да я запише там. Засега обаче няма да се впускаме в технически неща, а вместо това ще се фокусираме само върху алгоритмите за обработка на данни. Най-простите възможни алгоритми работят с еднокомпонентни променливи - например разгледаните по-горе int, float и string. Така можем да използваме Python като калкулатор, в който въвеждаме числа през **input** канала, след което ползваме стандартни математически операции +, -, *, / и ** (събиране, изваждане, умножение, деление и степенуване) и изкарваме отговор с команда **print**. Напишете например програма, в която потребителят въвежда последователно x и n, след което на екрана му се изписва стойността на $\sqrt[n]{x}$ като представите коренуването във вид на степен. За да не се налага да копираме едни и същи парчета код в различни програми, най-удобно е често повтарящи се операции да бъдат обособявани като функции, които могат след това да бъдат извиквани многократно с различни стойности на променливите. Ще дадем пример с функцията на две променливи $f : x, y \to x^y$

```
In [ ]: def power(x,y):
            return x**y
```

която илюстрира стандартния синтаксис в Python, макар в по-интересните случаи да има още няколко реда код преди командата **return**. Веднъж зададена, функцията може да бъде викана многократно, дефинирана е за всички числови типове и не работи само когато $x = 0$, а степента е отрицателна или комплексна.

Масиви, цикли и рекурсия

За да използваме пълноценно предимствата на персоналния компютър пред джобния калкулатор, ще се запознаем и с многокомпонентните типове данни, известни още като масиви. Двата най-прости и същевременно широко приложими вида в Python са списъците (list) и множествата (set). Списъкът е индексиран масив от данни, който се задава така x = [a, b, c, ...] като елементите са номерирани от 0 до $n-1$ където n = **len**(x) се нарича дължина на списъка (брой елементи). Така в горния пример можем да извикаме първия елемент като x[0], втория като x[1] и т.н. Празният списък се задава с x = [ ] или x = **list**(), като към него след това можем да добавяме последователно елементи с x.**append**(), например x.**append**(0) ще трансформира списъка в x = [0], а ако след това приложим x.**append**(1), ще имаме съответно x = [0, 1] и така нататък. За да се избегне повторението на еднотипни действия, в програмирането се използват цикли - с тях даваме указания на процесора да ги изпълнява вместо нас. Така например, ако искаме да попълним списъка до x = [0, 1, ... 9], можем да използваме цикъл

```
In [ ]: x = []
        for i in range(10):
            x.append(i)
        print(x)
```

който ще ни даде желания резултат. Поредицата от операции след двуеточието, изброени на нов ред, като се спазва правилното отстояние от четири интервала, се изпълняват за всяка стойност на индекса i от 0 (което е долната граница на **range** по подразбиране, ако не е указана друга) до най-високата стойност, без последната да се включва. Командата **print** се изпълнява само веднъж, след завършване на цикъла, тъй като тя не е написана на съответното отстояние - ако я преместим четири интервала надясно, при изпълнение на програмата ще виждаме всички междинни резултати на екрана си. След като вече имаме кода, лесно можем да разширим обхвата да кажем до **range**(100), или пък да променим правилото, по което се попълва съответният списък, например с x.append(i**2) което ще ни даде квадратите на всички едноцифрени числа. Ако по някаква причина ни трябват и двата списъка, лесно можем да изобразим единия в другия с операция, която в Python се нарича *list comprehension* и има простия синтаксис

y = [xi**2 for xi in x]

Това автоматично попълва списъка y с квадратите на елементите на x. Дори без да сме дефинирали предварително x, лесно можем да създадем y и с командата

y = [i**2 for i in **range**(10)]

която върши същата работа. Още няколко полезни операции: x.**insert**(k,*var*) вмъква елемента *var* на k-та позиция (винаги се брои от нула), а x.**remove**(*var*)

премахва първия елемент на x със стойност *var*. Също така x.**sort**() сортира елементите на x по големина (и/или азбучен ред), а x.**reverse**() пък обръща реда им.

Един малко по-интересен пример е редицата $n!$, която можем да зададем с рекурентната зависимост $a_n = na_{n-1}$ и началното условие $a_0 = 1$. Директното решение използва рекурсия, т.е. функцията вика сама себе си още в самата дефиниция

```
In [ ]: def fact(n):
            if n==0:
                return 1
            else:
                return n*fact(n-1)
```

което следва съвсем буквално логиката при рекурентното задаване на редицата. Така за да пресметне елемента a_n, алгоритъмът се връща чак до a_0 и възстановява един по един всички елементи до n-тия. Това обаче не е особено ефективно ако ни се налага да ползваме функцията многократно, тъй като междинната информация се забравя и всеки път се преизчислява наново. Далеч по-разумно е всички получени стойности да се съхраняват в списък, достъпен за потребителя

```
In [ ]: def fact(n):
            x = [1]
            for i in range(1,n+1):
                x.append(i*x[i-1])
            return x
```

като отново започваме с $x = [a_0]$, но попълваме един по един всички елементи до n включително (затова в **range** имаме горна граница n-1). Ако искаме да извикаме само последната стойност n!, това може да стане с командата **fact**(n)$[-1]$, която посочва последния елемент в списъка, аналогично x$[-2]$ дава предпоследни и т.н.

По същия начин ще предложим два варианта за пресмятане на биномните коефициенти $\binom{n}{k}$, които познаваме от комбинаториката. Първото решение ползва чиста рекурсия, базираща се на известното от триъгълника на Pascal свойство

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{0} = \binom{n}{n} = 1.$$

Следвайки тази логика съвсем буквално, стигаме до функция от вида

```
In [ ]: def binom(n,k):
            if k==0 or k==n:
                return 1
            else:
                return binom(n-1,k-1)+binom(n-1,k)
```

Това обаче води до множество излишни пресмятания, което значително забавя времето за изпълнение, а може да препълни и буферната памет. От друга страна

```
In [ ]: def binom(n):
            b=[1]
            for i in range(1,n+1):
                if i > n/2:
                    b.append(b[n-i])
                else:
                    b.append(round(b[-1]*(n-i+1)/i))
            return b
```

е оптимизирана така, че да не пресмята едни и същи коефициенти многократно, а вместо това ги съхранява в масив, освен това съкращава предварително еднаквите множители в числителя и знаменателя, и ползва симетрията $C_n^k = C_n^{n-k}$. Сравнете двете решения при големи стойности на n и k, и напишете код, който да пресмята по подобен начин вариациите V_n^k в максимално оптимизиран вид. Въпреки горните примери, рекурсията има своите приложения и често дава елегантни решения там, където циклите се провалят, трябва обаче да се държи сметка преди всичко за ефективността - естетиката тук идва чак на второ място.

Ще кажем няколко думи и за множествата - макар те по принцип да се ползват по-рядко от списъците, имат широко приложение в контекста на настоящия курс. За разлика от списъка, този тип представлява ненаредена структура без повтарящи се елементи. Празното множество се задава като x = **set**() или просто x = { }, а добавянето на елемет *var* става с x.**add**(*var*), при това ако този елемент вече е наличен в x, нищо няма да се промени. Функцията **set** освен това превръща списъци в множества, като заличава наредбата и отъждествява еднаквите елементи, докато **list** от своя страна може да направи от едно множество списък, като въведе автоматична наредба. Тук имаме на разположение оператори за намиране на обединението, сечението и разликата на две множества x и y, като синтаксисът е съответно x.**union**(y), x.**intersection**(y) и x.**difference**(y), а x.**remove**(*var*) работи както при списъци, с тази разлика, че тук премахнатият елемент *var* е единствен. Командите лесно се забравят, но затова пък и лесно се намират в интернет или чрез функциите **help** и **dir**, освен това Python е гъвкав език и позволява безпроблемно излизане от ситуацията с помощта на списъци: например вместо x.**union**(y), бихме могли да ползваме **set**(**list**(x)+**list**(y)), докато x.**intersection**(y) е еквивалентно на {xi **for** xi **in** x **if** xi **in** y} по аналогия с конструкцията *list comprehension*, за която стана дума по-горе. Една интересна задача е да се опише множеството от всички подмножества на дадено множество x, или т. нар. *степенно множество* (power set), което стандартно се бележи с pow(x). Това е йерархична конструкция, позната в математиката като *решетка*: на дъното ѝ седи празното множество, след това на първото ниво се разполагат всички елементи на x, интерпретирани като множества, на второто - подмножествата от два елемента и така нататък, до като не стигнем до самото x, погледнато

като подмножество на себе си. Задачата, както се вижда, е чисто комбинаторна, но практическото и осъществяване в код си има някои тънкости - опитайте се да я решите (с рекурсия или динамично, както в примера с биномните коефициенти).

Завършваме с още една много полезна конструкция в програмирането - цикъла **while**. Синтаксисът е доста близък до този на цикъла **for**, но тук идеята е списъкът от команди следващи **while** A: да се изпълнява докато условието A е в сила. По тази причина трябва специално да се постараем от една страна условието да подлежи на проверка и от друга, след краен брой цикли да престане да бъде валидно. Това са най-типичните грешки при цикъла **while**, които няма как да възникват в цикъла **for**. Пример за първата е да се опитаме да сравним променливи от различен тип (като текст и число) или да извикаме елемент от даден списък с индекс, надвишаващ размерността му (като x[21] при **len**(x) = 20). В тези случаи бихме получили съобщение за грешка. Вторият тип гафове е малко по-неприятен, тъй като карат програмата да зацикли. Например алгоритъмът

```
In [ ]: s = 0
        k = 0
        while s < 1000:
            k += 1
            s += k**(1/2)
        print(k)
```

изчислява максималната стойност на k, за която сумата от квадратните корени на първите k естествени числа не надвишава 1000. Началната стойност на сумата s и индекса k е 0, след което ги попълваме вътре в цикъла: означението k += 1 еквивалентно на k = k + 1. Ако не повишаваме стойността на индекса на всяка стъпка (което в цикъла **for** става автоматично), условието s < 1000 няма да бъде нарушено и кодът ще се изпълнява докато не го прекратим с външна намеса. Друга възможна грешка е да не зададем начална стойност на s, тогава условието просто няма как да бъде проверено. Циклите **while** имат широко приложение в числените алгоритми за решаване на задачи с последователни приближения. При това, всеки цикъл **for** може да бъде заместен от цикъл **while**, но обратното изобщо не е вярно, както се вижда дори и от простия пример, приведен по-горе.

Задачи

1. Напишете функция, която пресмята елементите в редицата на Fibonacci: $a_{n+1} = a_n + a_{n-1}$ като $a_0 = a_1 = 1$ до зададена отнапред стойност на индекса.
2. Функция за вероятността m числа в една колка на тото 'k от n' да са верни.
3. Напишете функция, която изчислява вероятността при провеждане на n експеримента с два възможни изхода, в k от случаите да се наблюдава първият, който е с единична вероятност p. Да се изписва специфично съобщение за грешка ако въведените стойности за k, n или p не са допустими.