Dekoratorlarni tushunish har qanday Python dasturchisi uchun muhim bosqichdir. Ushbu maqola dekorativlar sizga qanday qilib samaraliroq va samarali Python dasturchisi bo'lishingizga yordam berishi haqida bosqichma-bosqich qo'llanmadir.

Python-dagi dekorativlar chaqiriladigan ob'ektning o'zini doimiy ravishda o'zgartirmasdan, *chaqiriladigan* ob'ektlarning (funktsiyalar, usullar va sinflar) xatti-harakatlarini kengaytirish va o'zgartirish imkonini beradi .

Mavjud sinf yoki funktsiyaning xatti-harakatlariga "biriktirilishi" mumkin bo'lgan har qanday etarlicha umumiy funksionallik dekorativlardan foydalanishning ajoyib namunasidir. Bunga quyidagilar kiradi:

* ro'yxatga olish,
* kirish nazorati va autentifikatsiyani ta'minlash,
* vaqtni boshqarish vositalari va funktsiyalari,
* Tezlik chegarasi,
* keshlash va boshqalar.

**Nima uchun Pythonda dekorativlarni o'rganishingiz kerak?**

Bu adolatli savol. Men hozir gapirgan narsa mavhum bo'lib tuyuladi va dekorativlar Python dasturchisiga kundalik ishida qanday yordam berishini tushunish qiyin bo'lishi mumkin. Mana bir misol:

Tasavvur qiling, sizning hisobot dasturingiz biznes mantig'iga ega 30 ta funktsiyaga ega. Yomg'irli dushanba kuni ertalab menejeringiz sizga aytadi:

“TPS hisobotlarini eslaysizmi? Hisobot ishlab chiqaruvchining har bir bosqichida kirish va chiqish ma'lumotlarini jurnalga qo'shishimiz kerak. XYZ kompaniyasiga bu audit uchun kerak. Men ularga buni chorshanbagacha hal qilishimiz mumkinligini aytdim.

Python dekorativlari bilan qanchalik tanish ekanligingizga qarab, bu so'rov sizning qon bosimingizni oshiradi yoki ushbu yangi talablarni asta-sekin qabul qiladi.

Dekorator haqida ma'lumotga ega bo'lmasangiz, ehtimol siz keyingi uch kun davomida ushbu 30 ta funktsiyaning har birini o'zgartirishga va ularni qo'lda ro'yxatdan o'tish qo'ng'iroqlari bilan aralashtirib yuborishga harakat qilasiz. Umuman olganda, vaqtingizni qiziqarli o'tkazing.

Agar siz dekoratorlarni bilsangiz, xotirjam jilmayib, "Yaxshi, bugun soat 14:00 da tayyor bo'ladi" kabi bir narsa deysiz.

[Shundan so'ng, siz umumiy @audit\_log](https://habr.com/users/audit_log) dekoratorining kodini chop etasiz (faqat 10 qator uzunlikda) va uni har bir funktsiya ta'rifidan oldin qo'shasiz. Keyin siz va'da qilasiz va bir chashka qahva ichasiz.

Bu yerda men bo‘rttirib aytyapman, albatta. Ammo ozgina. Dekoratorlar haqiqatan ham kuchli bo'lishi mumkin.

Men dekorativlarni tushunish har qanday tajribali Python dasturchisi uchun muhim bosqich deb aytardim. Ular bir nechta ilg'or til tushunchalarini, shu jumladan birinchi darajali funktsiyalarning xususiyatlarini yaxshi tushunishni talab qiladi.

Lekin:

**Dekoratorlarni tushunish bunga arziydi**

Python-da dekorativlarning qanday ishlashini tushunish juda foydali.

Albatta, dekorativlarni birinchi marta taqdim etilganda tushunish juda qiyin bo'lib tuyuladi, lekin bu juda foydali xususiyat bo'lib, uni uchinchi tomon ramkalari va Python standart kutubxonasida tez-tez ko'rishingiz mumkin.

Dekoratorlarni tushuntirish har qanday yaxshi Python o'quv qo'llanmasining muhim bo'limidir. Ushbu maqolada men sizni ular bilan bosqichma-bosqich tanishtirishga harakat qilaman.

Mavzuga kirishdan oldin, keling, Python-da birinchi darajali funktsiyalarning xususiyatlarini ko'rib chiqaylik. Men [dbader.org](https://dbader.org/blog/python-first-class-functions) saytida ularga qo'llanma yozdim , uni o'qish uchun bir necha daqiqa vaqt ajratishingizni tavsiya qilaman. Bu erda dekorativlarni tushunish uchun "birinchi darajali funktsiyalar" dan eng muhim xulosalar:

* Funktsiyalar ob'ektlardir - ular o'zgaruvchilarga tayinlanishi, boshqa funktsiyalarga o'tkazilishi va qaytarilishi mumkin.
* Funktsiyalar boshqa funktsiyalar ichida aniqlanishi mumkin va bola funktsiyasi ota-ona funksiyasining mahalliy holatini (leksik yopilishlar) olishi mumkin.

Xo'sh, ketishga tayyormisiz? Keling, boshlaymiz.

Python dekorator asoslari

Xo'sh, aslida dekorativlar nima? Ular boshqa funktsiyani "bezatadi" yoki "o'radi" va kodni o'ralgan funksiya bajarilishidan oldin va keyin bajarishga imkon beradi.

Dekoratorlar boshqa funktsiyalarning harakatlarini o'zgartirishi yoki kengaytirishi mumkin bo'lgan qayta ishlatiladigan modullarni aniqlash imkonini beradi. Bundan tashqari, ular sizga o'ralgan funktsiyani doimiy ravishda o'zgartirmasdan buni amalga oshirishga imkon beradi. Funktsiyaning harakati faqat bezatilganida *o'zgaradi* .

Xo'sh, oddiy dekoratorni amalga oshirish nimaga o'xshaydi? Umuman olganda, dekorator chaqiriladigan ob'ekt bo'lib, u qo'ng'iroq qilinadigan ob'ektni kiritadi va boshqa chaqiriladigan ob'ektni qaytaradi.

Quyidagi funktsiya bu xususiyatga ega va siz yozishingiz mumkin bo'lgan eng oddiy dekorativ deb hisoblanishi mumkin:

**def** null\_decorator(func):

**return** func

Ko'rib turganingizdek, null\_decoratorchaqiriladigan ob'ekt bo'lib, u boshqa chaqiriladigan ob'ektni kiritish sifatida qabul qiladi va uni o'zgartirmasdan bir xil kirish ob'ektini qaytaradi.

Keling, uni boshqa funktsiyani bezash (yoki o'rash) uchun ishlatamiz:

**def** greet():

**return** 'Hello!'

greet = null\_decorator(greet)

>>> greet()

'Hello!'

Ushbu misolda greetmen funktsiyani belgilab qo'ydim va keyin uni funksiya orqali ishga tushirish orqali darhol bezadim null\_decorator. Bilaman, bu hozircha unchalik foydali ko'rinmaydi (biz maxsus null dekoratorni foydasiz qilib ishlab chiqdik, to'g'rimi?), lekin bir muncha vaqt o'tgach, bu Pythonda dekorator sintaksisi qanday ishlashini aniqroq qiladi.

O'zgaruvchini aniq chaqirish va keyin uni qayta tayinlash null\_decoratoro'rniga , funktsiyani bir bosqichda bezash uchun Python sintaksisidan foydalanishingiz mumkin :greetgreet@

@null\_decorator

**def** greet():

**return** 'Hello!'

>>> greet()

'Hello!'

[Funksiya taʼrifidan oldin @null\_decorator](https://habr.com/users/null_decorator) satrlarini qoʻyish avval funksiyani belgilash va keyin unga dekoratorni qoʻllash bilan bir xil. Sintaksisdan foydalanish @oddiygina sintaktik shakar va bu tez-tez ishlatiladigan naqsh uchun stenografiyadir.

E'tibor bering, sintaksisdan foydalanish @funktsiyani to'g'ridan-to'g'ri belgilash vaqtida bezatadi. Bu mo'rt xakerlarsiz bezaksiz asl nusxaga kirishni qiyinlashtiradi. Shuning uchun, bezaksiz funksiyani chaqirish imkoniyatini saqlab qolish uchun siz ba'zi funktsiyalarni qo'lda bezashingiz mumkin.

Hozirgacha juda yaxshi. Keling, bu qanday amalga oshirilganini ko'rib chiqaylik.

**Dekoratorlar xatti-harakatni o'zgartirishi mumkin**

Endi siz dekorator sintaksisi bilan bir oz tanish bo'lganingizdan so'ng, keling, *aslida* biror narsani bajaradigan va bezatilgan funksiyaning harakatini o'zgartiradigan boshqa dekorator yozamiz.

Mana, bezatilgan funktsiya natijasini katta harflarga o'zgartiradigan biroz murakkabroq dekorator:

**def** uppercase(func):

**def** wrapper():

original\_result = func()

modified\_result = original\_result.upper()

**return** modified\_result

**return** wrapper

Kirish funksiyasini null dekorator qilganidek oddiygina qaytarish o‘rniga, dekorator uppercaseyangi funksiyani (yopish) aniqlaydi va undan kirish funksiyasini chaqirish vaqtida uning harakatini o‘zgartirish uchun o‘rash uchun ishlatadi.

Yopish wrapperbezaksiz kirish funksiyasiga kirish huquqiga ega va kirish funksiyasi chaqirilishidan oldin va keyin qo'shimcha kodni bajarishi mumkin. (Texnik jihatdan kiritish funksiyasini umuman chaqirish shart emas).

E'tibor bering, bezatilgan funktsiya ilgari hech qachon bajarilmagan. Aslida, bu nuqtada kirish funktsiyasini chaqirish hech qanday ma'noga ega emas - dekorator chaqirilganda o'zining kiritish funktsiyasining harakatini o'zgartirishi kerak.

uppercaseDekoratorning harakatini ko'rish vaqti keldi . Agar asl funktsiyani u bilan bezasangiz nima bo'ladi greet?

@uppercase

**def** greet():

**return** 'Hello!'

>>> greet()

'HELLO!'

Umid qilamanki, bu siz kutgan natija bo'ldi. Keling, bu erda nima sodir bo'lganini batafsil ko'rib chiqaylik. dan farqli o'laroq null\_decorator, dekorator funksiyani bezashda *boshqa funksiya ob'ektini*uppercase qaytaradi :

>>> greet

<function greet at 0x10e9f0950>

>>> null\_decorator(greet)

<function greet at 0x10e9f0950>

>>> uppercase(greet)

<function uppercase.<locals>.wrapper at 0x10da02f28>

Avval ko'rganingizdek, bu bezatilgan funksiya chaqirilganda uning harakatini o'zgartirish uchun kerak. Katta harf dekoratorining o'zi funksiyadir. Va u bezatgan kirish funktsiyasining "kelajakdagi xatti-harakatiga" ta'sir qilishning yagona yo'li - kirish funktsiyasini yopish bilan almashtirish (yoki o'rash).

Shuning uchun uppercaseu keyinroq chaqirilishi mumkin bo'lgan boshqa funktsiyani (yopish) belgilaydi va qaytaradi, dastlabki kiritish funktsiyasini ishga tushiradi va uning chiqishini o'zgartiradi.

Dekoratorlar qo'ng'iroq qilinadigan ob'ektning harakatini o'ram yordamida o'zgartiradilar, shuning uchun siz doimo asl nusxani o'zgartirishingiz shart emas. Chaqiriladigan ob'ekt doimiy o'zgarishlarga duch kelmaydi - uning xatti-harakati faqat bezatilganida o'zgaradi.

Bu sizga mavjud funktsiyalar va sinflarni jurnallar va boshqa vositalar kabi qayta foydalanish mumkin bo'lgan modullar bilan "biriktirish" imkonini beradi. Bu dekorativlarni Python-da ko'pincha standart kutubxona va uchinchi tomon paketlarida ishlatiladigan kuchli xususiyatga aylantiradi.

Qisqa tanaffus

Aytgancha, agar siz hozirgi vaqtda qisqa kofe tanaffusga muhtoj ekanligingizni his qilsangiz, bu mutlaqo normaldir. Menimcha, yopilishlar va dekorativlar Pythonda tushunish eng qiyin tushunchalardan biridir. Shoshilmang va birdaniga hamma narsani tushunishga urinmang. Tarjimon sessiyasida birin-ketin misollar keltirish ko'pincha mazmunni tushunishga yordam beradi.

Bilaman, hammasi siz uchun yaxshi bo'ladi :)

Bitta funktsiyaga bir nechta dekorativlarni qo'llash

Funktsiyaga bir nechta dekorator qo'llanilishi ajablanarli emas. Bu ularning effektlarini to'playdi va dekorativlarni qayta foydalanish mumkin bo'lgan modullar kabi foydali qiladi.

Mana bir misol. Keyingi ikkita dekorator bezatilgan funksiyaning chiqish satrini HTML teglari bilan o‘rab oladi. Teglar qanday joylashtirilganiga qarab, Python bir nechta dekorativlarni qo'llash tartibini ko'rishingiz mumkin:

**def** strong(func):

**def** wrapper():

**return** '<strong>' + func() + '</strong>'

**return** wrapper

**def** emphasis(func):

**def** wrapper():

**return** '<em>' + func() + '</em>'

**return** wrapper

Keling, ushbu ikkita dekoratorni olib, ularni greetbir vaqtning o'zida bizning funktsiyamizga qo'llaymiz. Buni amalga oshirish uchun siz odatiy sintaksisdan foydalanishingiz @va bitta funktsiya ustiga bir nechta dekorativlarni joylashtirishingiz mumkin:

@strong

@emphasis

**def** greet():

**return** 'Hello!'

Bezatilgan funktsiyani ishga tushirsangiz, qanday natija ko'rishni kutasiz? [@emphasis](https://habr.com/users/emphasis) dekoratori avval o'z tegini qo'shadimi yoki <em>@strong [ustunlik](https://habr.com/users/strong)   qiladimi? Bezatilgan funktsiyani chaqirganingizda nima sodir bo'ladi:

>>> greet()

'<strong><em>Hello!</em></strong>'

Bu erda siz dekorativlar qanday tartibda ishlatilganligini aniq ko'rishingiz mumkin: *pastdan yuqoriga* . Dastlab kiritish funksiyasi [@emphasis](https://habr.com/users/emphasis) decorator bilan o'ralgan , so'ngra hosil bo'lgan (bezatilgan) funksiya yana [@strong](https://habr.com/users/strong) decorator bilan o'ralgan .

Ushbu pastdan yuqoriga tartibni eslash uchun men bu xatti-harakatni "dekorator to'plami" deb atashni yaxshi ko'raman. Siz stackni pastki qismdan qurishni boshlaysiz va keyin yuqoriga ko'tarilish uchun yangi bloklarni qo'shishda davom etasiz.

Agar biz yuqoridagi misolni buzsak va dekorativlarni qo'llash uchun sintaksisdan foydalanmasak @, dekorativ funktsiyalarga qo'ng'iroqlar zanjiri quyidagicha ko'rinadi:

decorated\_greet = strong(emphasis(greet))

Bu erda yana dekoratorning birinchi qo'llanilishini ko'rishingiz mumkin emphasis, keyin esa natijada o'ralgan funksiya yana dekoratorga o'ralgan strong.

Bu shuningdek, dekorativ qoplamaning chuqur darajalari oxir-oqibat ishlashga ta'sir qiladi, chunki ular ichki funksiya chaqiruvlarini qo'shishda davom etadilar. Amalda bu odatda muammo emas, lekin agar siz unumdorlikni talab qiluvchi kod ustida ishlayotgan bo'lsangiz, buni yodda tutish kerak.

Argumentlarni qabul qiladigan dekoratsiya funktsiyalari

greetHozirgacha berilgan barcha misollar hech qanday dalil talab qilmaydigan oddiy nullary funktsiyani bezatadi . Shunday qilib, siz bu erda ko'rgan dekoratorlar kiritish funktsiyasiga argumentlarni uzatish bilan shug'ullanmagan.

Agar siz ushbu dekoratorlardan birini argumentlarni qabul qiluvchi funktsiyaga qo'llamoqchi bo'lsangiz, u to'g'ri ishlamaydi. O'zboshimchalik bilan argumentlar oladigan funktsiyani qanday bezash kerak?

Bu erda Python funksiyasi o'zgaruvchan sonli argumentlar bilan ishlashda \*args*yordam* beradi . \*\*kwargs Dekorator proxyushbu xususiyatdan foydalanadi:

**def** proxy(func):

**def** wrapper(\*args, \*\*kwargs):

**return** func(\*args, \*\*kwargs)

**return** wrapper

Ushbu dekorativning ikkita diqqatga sazovor tomoni bor:

* U barcha pozitsion va kalit argumentlarni yig'ish va ularni o'zgaruvchilarda ( va ) saqlash uchun yopish ta'rifidagi \*va operatorlaridan foydalanadi .\*\*wrapperargskwargs
* \*Keyin oʻramni yopish “argumentni ochish” va yordamida toʻplangan argumentlarni asl kiritish funksiyasiga oʻtkazadi \*\*.

(Yulduzcha va qoʻsh yulduzcha operatorlarining maʼnosi haddan tashqari yuklanganligi va ular qoʻllanilgan kontekstga qarab oʻzgarib turishi biroz uyatli. Lekin bu fikrni tushunasiz degan umiddaman).

Keling, dekorativning orqasidagi texnikani proxyfoydaliroq amaliy misol bilan kengaytiraylik. Mana, tracefunksiyaning argumentlarini va ishlayotgan natijalarini chop etadigan dekorator:

**def** trace(func):

**def** wrapper(\*args, \*\*kwargs):

print(f'TRACE: calling {func.\_\_name\_\_}() '

f'with {args}, {kwargs}')

original\_result = func(\*args, \*\*kwargs)

print(f'TRACE: {func.\_\_name\_\_}() '

f'returned {original\_result!r}')

**return** original\_result

**return** wrapper

Funksiyani bezash traceva uni chaqirish orqali siz bezatilgan funksiyaga uzatilgan argumentlarni va uning qaytish qiymatini chop etishingiz mumkin. Bu hali ham o'yinchoq misolidir, ammo bu nosozliklarni tuzatish uchun ajoyib yordam bo'ladi:

@trace

**def** say(name, line):

**return** f'{name}: {line}'

>>> say('Jane', 'Hello, World')

'TRACE: calling say() with ("Jane", "Hello, World"), {}'

'TRACE: say() returned "Jane: Hello, World"'

'Jane: Hello, World'

Nosozliklarni tuzatish haqida gapiradigan bo'lsak, dekorativlarni tuzatishda bir nechta narsalarni yodda tutish kerak.

"Debuggable" dekorativlarni qanday yozish kerak

Dekoratordan foydalansangiz, aslida bir funktsiyani boshqasiga almashtirasiz. Bu jarayonning kamchiliklaridan biri shundaki, u asl (bezaksiz) funksiyaga biriktirilgan metama’lumotlarning bir qismini “yashiradi”.

Masalan, asl funktsiya nomi, uning docstring va parametrlar ro'yxati yopish orqali yashiriladi:

**def** greet():

"""Return a friendly greeting."""

**return** 'Hello!'

decorated\_greet = uppercase(greet)

Agar siz ushbu funksiyaning istalgan metamaʼlumotlariga kirishga harakat qilsangiz, uning oʻrniga oʻramning yopish metamaʼlumotlarini koʻrasiz:

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

>>> decorated\_greet.\_\_name\_\_

'wrapper'

>>> decorated\_greet.\_\_doc\_\_

**None**

Bu nosozliklarni tuzatish va Python tarjimoni bilan ishlashni noqulay va qiyinlashtiradi. Yaxshiyamki, buning tezkor yechimi bor: Python standart kutubxonasiga kiritilgan [functools.wraps dekoratori.](https://docs.python.org/3/library/functools.html" \l "functools.wraps)

functools.wrapsSiz o'zingizning dekoratorlaringizda dekoratsiya qilinmagan funksiyadan bezatuvchining yopilishiga yetim metama'lumotlarni nusxalash uchun foydalanishingiz mumkin . Mana bir misol:

**import** functools

**def** uppercase(func):

@functools.wraps(func)

**def** wrapper():

**return** func().upper()

**return** wrapper

Dekorator tomonidan qaytarilgan oʻramni yopishni qoʻllash functools.wrapshujjat qatori va kiritish funksiyasining boshqa metamaʼlumotlarini oʻrab oladi:

@uppercase

**def** greet():

"""Return a friendly greeting."""

**return** 'Hello!'

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

functools.wrapsO'zingiz yozgan barcha dekorativlardan foydalanishni tavsiya qilaman . Bu ko'p vaqtni talab qilmaydi va kelajakda disk raskadrovka paytida sizni (va boshqalarni) ko'p bosh og'rig'idan xalos qiladi.