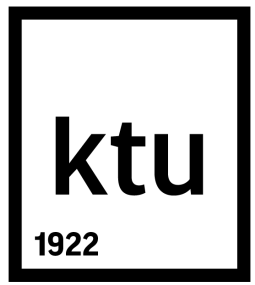
Lygiagretusis programavimas Python

Karolis Ryselis

Kauno Technologijos Universitetas



Q: Where do parallel functions wash their hands?
A: In Async

Paskaitos turinys

Python

- Python programavimo kalba sukurta 1991 m.
- Dabartinė versija 3.12 (2023 m. spalis)

Python

- Dinamiškai tipizuota kalba — tam pačiam kintamajam galima priskirti skirtingų tipų reikšmes (*duck typing*).
- Palaiko struktūrinį, objektinį, funkcinį programavimo stilius.
- Bendros paskirties kalba, daugiausia naudojama scenarijams, duomenų apdorojimui, tinklo sistemoms.
- Kalba yra interpretuojama, populiariausias interpretatorius — CPython.
- Vietoj riestinių skliaustų naudojamos įtraukos.
- Yra tik *foreach* stiliaus ir *while* ciklai.

Python sintaksė

- Funkcijos ir metodai apibrėžiami raktiniu žodžiu `def`, grąžinamos reikšmės ir parametrų tipų nurodyti nereikia:
  `def func(arg1, arg2):`
- Funkcijos kodas rašomas iš naujos eilutės atitraukus per 4 tarpus.
- Atitraukimas per 4 tarpus taikomas ir `if`, `for`, `while` ir kitiems sakiniams.
- Python leidžia iš funkcijos grąžinti keletą kintamųjų ir daryti priskyrimą į kelis kintamuosius.
- Neegzistuojančią reikšmę atitinka `None`, *bool* tipo reikšmės — `True` ir `False`.
- Duomenų struktūros — sąrašai (*list*), žodynai (*dict*), kortežai (*tuple*).
- *str* tipo kintamieji palaiko unikodą, *char* tipas neegzistuoja.

Python `while` ciklai

```
def get_fibonacci_number_at_index(index):
    a, b = 0, 1
    current_index = 0
    while current_index < index:
        a, b = b, a + b
        current_index += 1
    return a


fib_number = get_fibonacci_number_at_index(index=10)
print("10th Fibonacci number is",
      tenth_fibonacci_number)
```

Python sąlygos sakiniai

```
import platform


def get_current_os():
    platform_system = platform.system()
    if platform_system == "Linux":
        return "Superior OS"
    elif platform_system == "Windows":
        return "Inferior OS"
    else:
        return "Mediocre OS"


print(get_current_os())
```

Python *dict*

```
student_grades = {
    "Jonas": 4,
    "Petras": 9,
    "Antanas": 8
}

for name, grade in student_grades.items():
    print("Student", name, "got grade", grade)

highest_grade = max(student_grades.values())
print(highest_grade)
student_names = sorted(student_grades.keys())
for name in student_names:
    print(name)
```

Python *list comprehensions*

```
numbers = range(1, 21)
total = sum(i * i for i in numbers if i % 2 == 0)
print(total)
```

Namų užduotis

Python konsolėje įvykdyti šias eilutes:

```
import antigravity
```

```
import this
```

Python gijos

- Gijos Python atvaizduojamos `Thread` klase.
- Klase galima naudotis dviem būdais — paveldint klasę ir užklojant `run` metodą, arba kurti `Thread` objektą jam perduodant norimą vykdyti funkciją `target` parametru.
- Klasė turi klasikinius metodus:
  - `start()`
  - `join()`
  - `is_alive()`

Python užraktai

- Kritinės sekcijos apsauga realizuojama klase `Lock`.
- Pagrindiniai metodai — `acquire()` ir `release()`
- `Lock` objektu galima naudotis kaip konteksto tvarkytoju (*context manager*):

```
lock = Lock()
with lock:
    # this is run with mutual exclusion
```

Sąlyginė sinchronizacija

- Sąlygine sinchronizacija realizuojama klase `Condition`
- Pagrindiniai metodai:
  - `wait`
  - `wait_for`
  - `notify`
  - `notify_all`

Paprastas skaitiklis

```
class SimpleBoundedCounter:
    MIN = 0
    MAX = 50

    def __init__(self):
        self.count = self.MIN
        self._lock = Lock()
        self._condition = Condition(self._lock)

    def increase(self):
        with self._lock:
            self._condition.wait_for(lambda:self.count<self.MAX)
            self.count += 1
            self._condition.notify_all()

    def decrease(self):
        with self._lock:
            self._condition.wait_for(lambda:self.count>self.MIN)
            self.count -= 1
            self._condition.notify_all()
```

Paprastas skaitiklis

```
def increase(counter):
    for i in range(50):
        counter.increase()


def decrease(counter):
    for i in range(50):
        counter.decrease()
```

Paprastas skaitiklis

```
if __name__ == '__main__':
    simple_counter = SimpleBoundedCounter()
    threads = [Thread(target=increase, args=(simple_counter,))
               for i in range(10)]
    threads += [Thread(target=decrease, args=(simple_counter,))
                for i in range(9)]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
    print(simple_counter.count)
```

Globalus interpretatoriaus užraktas

- CPython interpretatorius turi globalų interpretatoriaus užraktą (*global interpreter lock (GIL)*), kuris apsaugo python objektus nuo lygiagrečios prieigos.
- *GIL* reikalingas, nes python atminties valdymas nėra saugus lygiagrečiai prieigai.
- Dėl *GIL* python programos yra vykdomos vienoje gijoje, o programos viduje sukurtos gijos yra vykdomos keičiant kontekstą vienoje CPU gijoje.
- Python realizacijos Jython (python JVM) ir IronPython (python .NET) neturi *GIL* ir gali išnaudoti visus procesoriaus branduolius, bet jos nėra plačiai naudojamos.
- Ši skaidrė gali pasenti kaip pienas – Python 3.13 planuojama turėti apėjimą globaliam interpretatoriaus užraktui.

Python *multiprocessing*

- Python gijos tinka tada, kai lygiagrečiai atliekami veiksmai yra daugiausiai įvestis/išvestis (*IO bound*).
- Python turi modulį *multiprocessing*, kuris leidžia kurti procesus. Kiekvienas procesas yra atskiras Python interpretatoriaus procesas, todėl jais naudojantis galima išnaudoti sistemos lygiagrečias galimybes.

`Process` klasė

- Klasė `Process` turi tuos pačius metodus, kaip `Thread` klasė, bet paleidžia naują procesą, ne giją.
- Kadangi funkcijos vykdomos skirtinguose procesuose, nebelieka bendros atminties.
- Python `multithreading` modulis veikia **bendros** atminties principu.
- Python `multiprocessing` modulis veikia **paskirstytos** atminties principu.

Apsikeitimas duomenimis tarp procesų

- Apsikeitimas duomenimis tarp procesų vykdomas tokiomis priemonėmis:
  - Eilėmis `Queue`;
  - Kanalais `Pipe`.

Duomenų perdavimas naudojant eiles

- Sukuriamas `multiprocessing.Queue` objektas, kurio sąsaja yra beveik identiška standartinės Python eilės sąsajai.
- Į eilę objektai įrašomi metodu `put`.
- Iš eilės objektai išimami metodu `get`.
- Vienas procesas į eilę rašo duomenis, kitas iš eilės duomenis ima.

Duomenų perdavimas naudojant eiles

```
def generate_items_to_queue(queue):
    for _ in range(40000):
        random_string = get_random_string()
        queue.put(random_string)
```

Duomenų perdavimas naudojant eiles

```
def main():
    queue = multiprocessing.Queue()
    cpu_count = multiprocessing.cpu_count()
    lock = multiprocessing.Lock()
    processing_processes = [multiprocessing.Process(
            target=process_elements,
            args=(queue, lock)
        )
        for _ in range(cpu_count)]
    for process in processing_processes:
        process.start()
    generate_items_to_queue(queue)
    for _ in range(cpu_count):
        queue.put(None)
    for process in processing_processes:
        process.join()
```

Duomenų perdavimas naudojant eiles

```
def process_elements(queue, lock):
    while True:
        item = queue.get()
        if item is None:
            break
        process_item(item, lock)
```

Duomenų perdavimas naudojant kanalus

- Kviečiama funkcija `multiprocessing.Pipe`, kuri sukuria du kanalo galus (`Connection` tipo) ir juos grąžina:
- `parent_conn, child_conn = Pipe()`
- Kanalo galais galima naudotis kviečiant jų metodus `send` ir `recv`.
- Komunikacija kanalu galima naudotis siunčiant ir gaunant žinutes iš bet kurio kanalo.
- Kanalai yra *One2One* tipo — bandymas į tą patį kanalą rašyti ar iš to paties kanalo skaityti iš kelių procesų vienu metu gali sugadinti siunčiamus duomenis.

Duomenų perdavimas naudojant kanalus

```
def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()


def main():
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()
```

Kritinė sekcija tarp procesų

- `multiprocessing` modulis turi tuos pačius sinchronizacijos primityvus, kaip ir `multithreading` modulis.
- Jie naudingi, kai keli procesai gali naudotis tuo pačiu resursu, pvz., rašyti į failą.

Procesų telkiniai

- Procesų telkinys (*process pool*) — fiksuoto dydžio procesų rinkinys, kuriam galima paskirstyti darbus.
- Python procesų telkiniai kuriami pasinaudojant `Pool` klase.

`Pool` klasė

`Pool(processes, initializer, initargs, maxtasksperchild, context)`

Sukuria procesų telkinį.

`processes` kiek procesų sukurti, jei nenurodyta, naudojama `os.cpu_count()` reikšmė.

`initializer` jei nurodyta, kiekvienas procesas sukūrimo metu iškvies `initializer(*initargs)`.

`maxtasksperchild` kiek užduočių gali įvykdyti procesas, kol bus pakeistas nauju procesu.

`context` procesų sukūrimo kontekstas.

`Pool` klasė

`apply(func, args, kwargs)`

Iškviečia funkciją `func` viename iš procesų telkinio procesų su parametrais `args`, `kwargs`.

`map(func, iterable)`

Iškviečia funkciją `func` kiekvienam `iterable` elementui darbą išdalinant procesams procesų telkinyje.

Funkcijos turi asinchronines versijas `apply_async` ir `map_async`, kurios grąžina `AsyncResult` objektus ir neblokuoja vykdymo.

Procesų telkiniai

```
def square(x):
    return (x * x) ** 17


numbers = list(range(100000))
with Pool() as pool:
    result = pool.map(square, numbers)
print(result)
```

Bendra atmintis tarp procesų

- Procesas gali sukurti atminties bloką, pasiekiamą iš kito proceso.
- C kalboje tam skirtos kelios funkcijos, pvz., `mmap`.
- `mmap` kviečiama kaskart kviečiant `malloc`, bet suteikia daugiau laisvės.
- Vienas `mmap` parametrų yra išskiriamos atminties tipas, galimos reikšmės — `MAP_SHARED` ir `MAP_PRIVATE`.
- Išskyrus atmintį su `MAP_SHARED` ji tampa prieinama visiems procesams.

Python bendra atmintis tarp procesų

- Python suteikia prieigą prie bendros procesų atminties naudojant `Value` ir `Array` objektus.
- `Value` yra skirtas vienam bendram kintamajam, `Array` — masyvui.
- Priešingai, nei įprastame Python kode, palaikomi tik standartiniai C tipai.

Bendra atmintis tarp procesų

```
items = [randint(0, MAX_NUMBER) for _ in range(MAX_ITEM)]
shared_array = Array('i', items)


def calculate_powers(index):
    shared_array[index] = shared_array[index] ** POWER


with Pool() as pool:
    pool.map(calculate_powers, range(MAX_ITEM))
for item in shared_array:
    print(item)
```

Bendra atmintis tarp procesų

- Naudoti ir paskirstytos, ir bendros atminties tarp procesų nerekomenduojama, nes tai apsunkina programos skaitomumą ir palaikomumą.
- Kartais bendra atmintis tarp procesų gali būti naudinga — nereikia siųsti duomenų tarp procesų, todėl galima gauti programos pagreitėjimą.

Serverio procesai

- Python turi klasę `SyncManager`, kuri gali valdyti bendrus tarp procesų duomenis.
- Naudojantis šia klase galima perduoti duomenis ir į kitą kompiuterį per tinklą.
- Ši klasė taip pat turi metodus barjerui, semaforams, užraktams kurti ir valdyti.
- Naudojantis serveriu procesų duomenys yra persiunčiami, tačiau nėra ribojimo, kokius tipus galima siuntinėti.

Serverio procesai

```
def parallel_sort(data: list[int], result_list: ListProxy):
    sorted_part = merge_sort(data)
    result_list.extend(sorted_part)

num_processes = 16
random_data = [random.randint(1, 1_000_000) for _ in range(750_000)]
with Manager() as manager:
    sorted_data = manager.list()
    processes = []
    for i in range(num_processes):
        start = i * (len(random_data) // num_processes)
        end = ((i + 1) * (len(random_data) // num_processes)
               if i < num_processes - 1 else len(random_data))
        process = Process(target=parallel_sort,
                          args=(random_data[start:end], sorted_data))
        processes.append(process)
        process.start()
    for process in processes:
        process.join()
    final_sorted_data = merge_sort(sorted_data)
```